PI 1159477

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office

April 28, 2004

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A FILING DATE.

APPLICATION NUMBER: *60/523,462*
FILING DATE: *November 18, 2003*
RELATED PCT APPLICATION NUMBER: *PCT/US04/03609*

By Authority of the
COMMISSIONER OF PATENTS AND TRADEMARKS

L. Wallace
T. WALLACE
Certifying Officer

PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

PTO/SB/16 (08-03)
Approved for use through 07/31/2006. OMB 0651-0032
U.S. Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

PROVISIONAL APPLICATION FOR PATENT COVER SHEET

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53(c).

Express Mail Label No. EL919127215US

17510 U.S.PTO 60/523462

INVENTOR(S)		
Given Name (first and middle [if any])	Family Name or Surname	Residence (City and either State or Foreign Country)
Aravind R. Sethuraman	Dasu Panchanathan	Tempe, Arizona Gilbert, Arizona

Additional inventors are being named on the 0 *separately numbered sheets attached hereto*

TITLE OF THE INVENTION (500 characters max)

A Methodology to Design a Dynamically Reconfigurable Processor

Direct all correspondence to: **CORRESPONDENCE ADDRESS**

☑ Customer Number: 28,529

OR

☐ Firm or Individual Name					
Address					
Address					
City		State		Zip	
Country		Telephone		Fax	

ENCLOSED APPLICATION PARTS *(check all that apply)*

☑ Specification *Number of Pages* 83
☐ CD(s), Number
☐ Drawing(s) *Number of Sheets*
☑ Other (specify) PowerPoint Presenta.
☐ Application Date Sheet. See 37 CFR 1.76

METHOD OF PAYMENT OF FILING FEES FOR THIS PROVISIONAL APPLICATION FOR PATENT

☑ Applicant claims small entity status. See 37 CFR 1.27.
☑ A check or money order is enclosed to cover the filing fees.
☑ The Director is herby authorized to charge filing fees or credit any overpayment to Deposit Account Number: 070135
☐ Payment by credit card. Form PTO-2038 is attached.

FILING FEE Amount ($): $80.00

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

☑ No.
☐ Yes, the name of the U.S. Government agency and the Government contract number are:

Respectfully submitted,

SIGNATURE

Date 11/18/03

REGISTRATION NO. 24,583 *(if appropriate)*

TYPED or PRINTED NAME Thomas D. MacBlain

Docket Number: 9138-0141

TELEPHONE 602-530-8088

USE ONLY FOR FILING A PROVISIONAL APPLICATION FOR PATENT

PTO/SB/17 (10-03)
Approved for use through 07/31/2006. OMB 0651-0032
U.S. Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

FEE TRANSMITTAL for FY 2004

Effective 10/01/2003. Patent fees are subject to annual revision.

[X] Applicant claims small entity status. See 37 CFR 1.27

TOTAL AMOUNT OF PAYMENT ($) 80.00

Complete if Known	
Application Number	
Filing Date	herewith
First Named Inventor	Dasu
Examiner Name	
Art Unit	
Attorney Docket No.	9138-0141

METHOD OF PAYMENT *(check all that apply)*

[X] Check [ ] Credit card [ ] Money Order [ ] Other [ ] None

[X] Deposit Account:

Deposit Account Number: 070135

Deposit Account Name: Gallagher & Kennedy, P.A.

The Director is authorized to: *(check all that apply)*

[ ] Charge fee(s) indicated below [X] Credit any overpayments

[X] Charge any additional fee(s) or any underpayment of fee(s)

[ ] Charge fee(s) indicated below, except for the filing fee to the above-identified deposit account.

FEE CALCULATION

1. BASIC FILING FEE

Large Entity Fee Code	Large Entity Fee ($)	Small Entity Fee Code	Small Entity Fee ($)	Fee Description	Fee Paid
1001	770	2001	385	Utility filing fee	
1002	340	2002	170	Design filing fee	
1003	530	2003	265	Plant filing fee	
1004	770	2004	385	Reissue filing fee	
1005	160	2005	80	Provisional filing fee	80

SUBTOTAL (1) ($) 80

2. EXTRA CLAIM FEES FOR UTILITY AND REISSUE

	Extra Claims		Fee from below		Fee Paid
Total Claims		-20** =		X	=
Independent Claims		- 3** =		X	=
Multiple Dependent					=

Large Entity Fee Code	Large Entity Fee ($)	Small Entity Fee Code	Small Entity Fee ($)	Fee Description
1202	18	2202	9	Claims in excess of 20
1201	86	2201	43	Independent claims in excess of 3
1203	290	2203	145	Multiple dependent claim, if not paid
1204	86	2204	43	** Reissue independent claims over original patent
1205	18	2205	9	** Reissue claims in excess of 20 and over original patent

SUBTOTAL (2) ($)

**or number previously paid, if greater; For Reissues, see above

FEE CALCULATION (continued)

3. ADDITIONAL FEES

Large Entity Fee Code	Large Entity Fee ($)	Small Entity Fee Code	Small Entity Fee ($)	Fee Description	Fee Paid
1051	130	2051	65	Surcharge - late filing fee or oath	
1052	50	2052	25	Surcharge - late provisional filing fee or cover sheet	
1053	130	1053	130	Non-English specification	
1812	2,520	1812	2,520	For filing a request for *ex parte* reexamination	
1804	920*	1804	920*	Requesting publication of SIR prior to Examiner action	
1805	1,840*	1805	1,840*	Requesting publication of SIR after Examiner action	
1251	110	2251	55	Extension for reply within first month	
1252	420	2252	210	Extension for reply within second month	
1253	950	2253	475	Extension for reply within third month	
1254	1,480	2254	740	Extension for reply within fourth month	
1255	2,010	2255	1,005	Extension for reply within fifth month	
1401	330	2401	165	Notice of Appeal	
1402	330	2402	165	Filing a brief in support of an appeal	
1403	290	2403	145	Request for oral hearing	
1451	1,510	1451	1,510	Petition to institute a public use proceeding	
1452	110	2452	55	Petition to revive - unavoidable	
1453	1,330	2453	665	Petition to revive - unintentional	
1501	1,330	2501	665	Utility issue fee (or reissue)	
1502	480	2502	240	Design issue fee	
1503	640	2503	320	Plant issue fee	
1460	130	1460	130	Petitions to the Commissioner	
1807	50	1807	50	Processing fee under 37 CFR 1.17(q)	
1806	180	1806	180	Submission of Information Disclosure Stmt	
8021	40	8021	40	Recording each patent assignment per property (times number of properties)	
1809	770	2809	385	Filing a submission after final rejection (37 CFR 1.129(a))	
1810	770	2810	385	For each additional invention to be examined (37 CFR 1.129(b))	
1801	770	2801	385	Request for Continued Examination (RCE)	
1802	900	1802	900	Request for expedited examination of a design application	

Other fee (specify) _____

*Reduced by Basic Filing Fee Paid SUBTOTAL (3) ($)

SUBMITTED BY				(Complete *(if applicable)*)	
Name (Print/Type)	Thomas D. MacBlain	Registration No. (Attorney/Agent)	24,583	Telephone	602-530-8088
Signature				Date	11/18/04

WARNING: Information on this form may become public. Credit card information should not be included on this form. Provide credit card information and authorization on PTO-2038.

This collection of information is required by 37 CFR 1.17 and 1.27. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 12 minutes to complete, including gathering, preparing, and submitting the completed application form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, P.O. Box 1450, Alexandria, VA 22313-1450. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. **SEND TO: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.**

If you need assistance in completing the form, call 1-800-PTO-9199 and select option 2.

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant: Dasu et al.

Filed: Herewith

Title: **A Methodology to Design a Dynamically Reconfigurable Processor**

CERTIFICATE OF MAILING BY EXPRESS MAIL
"Express Mail" mailing label number EL919127215US

Mail Stop Provisional Patent Application
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Commissioner:

I hereby certify that the following correspondence is being deposited in the United States Postal Service as Express Mail on the date shown below in an envelope addressed as shown above.

1. Provisional Application for Patent Cover Sheet (1 page);
2. Fee Transmittal for FY 2004 (1 page in duplicate);
3. Specification (83 pages);
4. Power Point Presentation (41 pages);
5. Check for $80.00; and
6. A return receipt postcard.

18 November 2003
Date

Suzanne Shields
Suzanne Shields

GALLAGHER & KENNEDY, P.A.
2575 East Camelback Road
Phoenix, Arizona 85016-9255
Tel. No. (602) 530-8000
Fax. No. (602) 530-8500

A Methodology to Design a Dynamically Reconfigurable Processor

PhD Proposal Document

By
Aravind R. Dasu
Department of Electrical Engineering

Advisor: Dr. S. Panchanathan

Table of Contents

Abstract

Multimedia processing with emphasis on graphics is becoming increasingly important with wide variety of applications ranging from multimedia cell phones to high definition interactive television. Media processing involves the capture, storage, manipulation and transmission of multimedia objects such as text, handwritten data, audio objects, still images, 2D/3D graphics, animation and full-motion video. A number of implementation strategies have been proposed for processing multimedia data. These approaches can be broadly classified based on the evolution of processing architectures and the functionality of the processors. In order to provide media processing solutions to different consumer markets, designers have combined some of the classical features from both the functional and evolution based classifications resulting in many hybrid solutions. Multimedia and Graphics applications are computationally intensive and have been traditionally solved in 3 different ways. One is through the use of a high speed general purpose processor with accelerator support, which is essentially a sequential machine with enhanced instruction set architecture. Here the overlaying software bears the burden of interpreting the application in terms of the limited tasks that the processor can execute (instructions) and schedule these instructions to avoid resource and data dependencies. The second is through the use of an Application Specific Integrated Circuit (ASIC) which is a completely hardware oriented approach, spatially exploiting parallelism to the maximum extent possible. The former, although slower, offers the benefit of hardware reuse for executing other applications. The latter, albeit faster and more power, area & time efficient for a specific application, offers poor hardware reutilization for other applications. The third is through specialized programmable processors such as DSPs and media processors. These attempt to incorporate the programmability of general purpose processors and provide some amount of spatial parallelism in their hardware architectures.

The complexity, variety of techniques and tools, and the high computation, storage and I/O bandwidths associated with multimedia processing presents opportunities for reconfigurable processing to enables features such as scalability, maximal resource utilization and real-time implementation. The relatively new domain of reconfigurable solutions lies in the region of computing space that offers the advantages of these approaches while minimizing their drawbacks. Field Programmable Gate Arrays (FPGAs) were the first attempts in this direction. But poor on-chip network architectures lead to high reconfiguration times and power consumptions. Improvements over this design using Hierarchical Network architectures with RAM style configuration loading have lead to a factor of 2-4 times reduction in individual configuration loading times. But the amount of redundant and repetitive configurations still remains high. This is one of the important factors that leads to the large overall configuration times and high power consumption compared to ASIC or embedded processor solutions. We believe that designing processing elements based on identifying correlated compute intensive regions within each application and between applications will result in large amounts of processing in localized regions of the chip. This reduces the amount of reconfigurations and hence faster application switching. This will also reduce the amount of on-chip communication, which in turn helps reduce power consumption. Since applications can be represented as Control Data Flow Graphs (CDFGs) such a pre-processing analysis lies in the area of pattern matching, specifically graph matching. In this context we propose a reduced complexity, yet exhaustive enough graph matching algorithm. We further propose to reduce the amount of on-chip communication by adopting reconfiguration aware static scheduling to manage task and resource dependencies on the processor. This is complemented by a divide and conquer approach which

helps in the allocation of an appropriate number of processing units aimed towards achieving uniform resource utilization.
To validate the success of this approach, we will obtain estimates of reconfiguration times and also show the potential for reduction in power consumptions, by performing the experiment of identifying correlated-compute intensive regions on an assorted set of algorithms taken from the media standards such as MPEG-4 and frequently used graphics algorithms.

1 Introduction

A variety of media processing techniques are typically used in multimedia processing environments to capture, store, manipulate and transmit multimedia objects such as text, handwritten data, audio objects, still images, 2D/3D graphics, animation and full-motion video. Example techniques include speech analysis and synthesis, character recognition, audio compression, graphics animation, 3D rendering, image enhancement and restoration, image/video analysis and editing, and video transmission. Multimedia computing presents challenges from the perspectives of both hardware and software. For example, multimedia standards such as MPEG-1, MPEG-2, MPEG-4, MPEG-7, H.263 and JPEG 2000 involve execution of complex media processing tasks in real-time. The need for real-time processing of complex algorithms is further accentuated by the increasing interest in 3-D image and stereoscopic video processing. Each media in a multimedia environment requires different processes, techniques, algorithms and hardware. The complexity, variety of techniques and tools, and the high computation, storage and I/O bandwidths associated with multimedia processing presents opportunities for reconfigurable processing to enables features such as scalability, maximal resource utilization and real-time implementation.
To demonstrate the potential for reconfiguration in multimedia computations, we have performed a detailed complexity analysis of the recent multimedia standard (MPEG-4) [80], which we believe involves multiple media and encompasses a wide range of operations typically found in media processing. The results of our analysis show that there are significant variations in the computational complexity among the various modes/operations of MPEG-4. This points to the potential for extensive opportunities for exploiting reconfigurable implementations of multimedia/ graphics algorithms.
The availability of large, fast, FPGAs is making possible reconfigurable implementations for a variety of applications. FPGAs consist of arrays of Configurable Logic Blocks (CLBs) that implement various logical functions. The latest FPGAs from vendors like Xilinx and Altera can be partially configured and run at several megahertz. Ultimately, computing devices may be able to adapt the underlying hardware dynamically in response to changes in the input data or processing environment and process real time applications. Thus FPGAs have established a point in the computing space which lies in between the dominant extremes of computing, ASICS and software programmable/ instruction set based architectures. There are three dominant features that differentiate reconfigurable architectures from instruction set based programmable computing architectures & ASICs: (i) spatial implementation of instructions through a network of processing elements with the absence of explicit instruction fetch-decode model (ii) flexible interconnects which support task dependent data flow between operations (iii) ability to change the Arithmetic and Logic functionality of the processing elements. The reprogrammable space is characterized by the allocation and structure of these resources. Computational tasks can be implemented on a reconfigurable device with intermediate data flowing from the generating function to the receiving function. The salient features of reconfigurable machines are:

- Instructions are implemented through locally configured processing elements, thus allowing the reconfigurable device to effectively process more instructions into active silicon in each cycle.
- Intermediate values are routed in parallel from producing functions to consuming functions (as space permits) rather than forcing all communication to take place through a central resource bottleneck.
- Memory and interconnect resources are distributed and are deployed based on need rather than being centralized, hence presenting opportunities to extract parallelism at various levels.

The networks connecting the Configuration Logic Blocks or Units (CLBs) or processing elements can range from full connectivity crossbar to neighbor only connecting mesh networks. The best characterization to date which empirically measures the growth in the interconnection requirements with respect to the number of Look-Up Tables (LUTs) is the Rent's rule which is given as follows:

$N^{io} = C N^{p}_{gates}$

Where N^{io} corresponds to the number of interconnections (in/out lines) in a region containing N_{gates}. C and p are empirical constants. For logical functions typically p ranges from $0.5<p<0.7$.

It has been shown [1] (by building the FPGA based on Rent's model and using a hierarchical approach) that the configuration instruction sizes in traditional FPGAs are higher than necessary, by at least a factor of 2-4. Therefore for rapid configuration, off-chip context loading becomes slow due to the large amount of configuration data that must be transferred across a limited bandwidth I/O path. It is also shown that greater word widths increase wiring requirements, while decreasing switching requirements. In addition, larger granularity data paths can be used to reduce instruction overheads. The utility of this optimization largely depends on the granularity of the data which needs to be processed. However, if the architectural granularity is larger than the task granularity, the device's computational power will be under utilized. Another promising development in efforts to reduce configuration time is shown in [2]. The authors propose the use of a random access technique to selectively load a configurable logic unit on the processor. This adds some overheads in terms of address decoding circuitry, but is very low compared to the rest of the routing resources on the chip. It is shown to be efficient compared to the shift register style of programming the processor. This approach still doesn't reduce the number of configurations that need to be performed for migrating from one portion of an application to another portion in the same application or a different one.

Most of the current approaches towards building a reconfigurable processor are targeted towards performance in terms of speed and are not tuned for power awareness or configuration time optimization. Therefore certain problems have surfaced that need to be addressed at the pre-processing phase.

Firstly, the granularity or the processing ability of the Configurable Logic Units (CLUs) must be driven by the set of applications that are intended to be ported onto the processing platform. Some research groups have taken the approach of visual inspection [3], while others have adopted algorithms of exponential complexity [4,5] to identify regions in the application's DFGs that qualify for CLUs. None of the current approaches attempt to identify the regions through an automated low complexity approach that deals with CDFGs.

Secondly, the number of levels in hierarchical network architecture must be influenced by the number of processing elements or CLUs needed to complete the task / application. This in turn depends on the amount of parallelism that can be extracted from the algorithm and the

percentage of resource utilization. To the best of our knowledge no research group in the area of reconfigurable computing has dealt with this problem.
Thirdly, the complex network on the chip, makes dynamic scheduling expensive as it adds to the primary burden of power dissipation through routing resource utilization. Therefore there is a need for a reconfiguration aware scheduling strategy. Most research groups have adopted dynamic scheduling for a reconfigurable accelerator unit through a scheduler that resides on a host processor [6,7].
The increasing demand for fast processing, high flexibility and reduced power consumption naturally demand the design and development of a low configuration time aware-dynamically reconfigurable processor.

2 Related Work

The literature review on related work is distributed into 2 phases. In the first phase we will discuss the approaches taken by other researchers in the design of reconfigurable processors. But the design methodology for a reconfigurable processor can be better analyzed and understood by breaking it down into smaller sub-problems which are linked together. Therefore we have distributed the second phase of the literature survey wherein relevant background work carried out by researchers in other fields of engineering (who share similar sub-problems) will be periodically provided for the smaller problems individually.

2.1 Review of literature on Reconfigurable Computing

Reconfigurable computing is a paradigm of computing which aims to achieve the performance of an ASIC with the flexibility that is associated with a General-purpose processor. ASICs offer the best individual hardware implementations, but lack flexibility to execute a variety of these algorithms. General purpose computing machines are highly flexible as any algorithm can be expressed in terms of the machine's language (instruction set). The concept of reconfigurable computing attempts to find a solution that has the advantages of both paradigms and has been well defined in [1,8]. A wide range of solutions varying from the Xilinx FPGA to HCS research laboratories' Dynamically Reconfigurable Network Processor have been proposed so far. Apart from the plethora of work being undertaken in the academia, various companies have also come up with commercial solutions. This section presents a survey of both commercially available reconfigurable solutions and those proposed by researchers in the academia.
Reconfigurable devices can be categorized as follows according to their properties.

1. Reconfiguration methodology
   - Run-time
   - Static
   - Hybrid
2. Granularity
   - Look Up Table based Field Programmable Gate Array
   - Configuration Logic Block
   - Functional Unit
3. Interconnect
   - Mesh
   - Hierarchical
   - Linear

- Cross-bar

4. Functionality
   - DSP (Digital Signal Processing)
   - Co-processor
   - Accelerator
   - Stand-alone
5. Application
   - General-purpose
   - Class of applications
   - Special-purpose

Criteria for classification of reconfigurable processors

a. Reconfiguration methodology

Reconfiguration methodology gives rise to three types of Reconfigurable Processors (RP): Statically reconfigurable, dynamically reconfigurable and hybrid. By hybrid, we refer to those processors which can be partially reconfigured at run-time.

b. Granularity of a reconfigurable processor

Granularity of a RP defines the types of applications that the RP can attempt to solve. This property along with the interconnect structure determines the power consumption of the RP to a great extent. A coarse-granular processor must be supported by an efficient routing architecture to keep the power consumption within bounds. Granularity can range from the fine-granular FPGAs to the coarse-granular Reconfigurable SOCs. Three levels of granularity can be identified: Look-up table based, Configurable Logic Blocks based, Functional unit based.

c. Interconnect structure of the reconfigurable processor

Interconnect structure is an important attribute of a reconfigurable processor. It has been shown that 70% of the energy consumed by a reconfigurable chip is due to interconnects. Interconnect architecture must serve two varying purposes: provide maximum connectivity between processing blocks and occupy minimal area of the circuitry. The number of switches along the various interconnect paths must be minimized, because this determines the time taken to reconfigure a path or circuit. Four standard interconnect architectures have been used widely. They are mesh, hierarchical, linear and crossbar. Some of the RP use a combination of two or more of these interconnect structures.

d. Functionality of a reconfigurable processor

This property determines how the RP is implemented in the overall architecture and defines the purpose of the RP. Parameters like the amount of resources available to the RP and the level of interactivity between these resources can be determined. The four types of RP based on this criteria are DSP, coprocessor, accelerator and stand-alone.

e. Target application domain of the reconfigurable processor

There are three types of target applications that existing RP have been designed for. Some processors have been designed to support general-purpose computing with reconfiguration being used to accelerate the execution process. A second type of processors targets the special-purpose domain, wherein they try to gain an enhancement in performance with the help of reconfiguration. A third type is targeted towards a class of applications wherein the applications

have a large number of kernels in common. Reconfiguration can be used to switch applications in this class.

Table 1: Classification of Reconfigurable Processors (RP)

Processor	Reconfiguration method	Granularity	Interconnect	Functionality	Application
Piperench	Hybrid	CLB	Linear	Stand-alone	Special-purpose
FIPSOC	Hybrid	CLB	Mesh	Co-processor	Class of applications
Chameleon	Hybrid	FU	Hierarchical	Co-processor	Class of applications
Matrix	Hybrid	FU	Hierarchical	Stand-alone	General-purpose
CSoC	Hybrid	FU	Hierarchical	Co-processor	Class of applications
RaPiD	Hybrid	FU	Linear	DSP	Class of applications
Garp	Run-time	CLB	Crossbar	Co-processor	General-purpose
MorphoSys	Run-time	CLB	Mesh	Stand-alone	General-purpose
SOP	Run-time	CLB	Mesh	Co-processor	General-purpose
Chimaera	Run-time	CLB	Mesh	DSP	General-purpose
Plado	Run-time	FPGA	Other	Co-processor	General-purpose
PRISM	Run-time	FPGA	Other	Stand-alone	General-purpose
Compare	Run-time	FPGA	Crossbar	DSP	General-purpose
CHESS Array	Run-time	FPGA	Mesh	Stand-alone	General-purpose
Spyder	Run-time	FU	Other	DSP	General-purpose
Concise	Run-time	FU	Other	DSP	General-purpose
Raw Architecture Workstation	Run-time	FU	Mesh	Stand-alone	General-purpose
KressArray	Run-time	FU	Mesh	Stand-alone	Special-purpose
Colt	Run-time	FU	Mesh	Stand-alone	General-purpose
ReMarc	Run-time	FU	Mesh	Co-processor	Special-purpose
DreAM	Run-time	FU	Mesh	Co-processor	Class of applications
CALISTO	Static	CLB	Other	Accelerator	Class of applications
Low energy FPGA	Static	FPGA	Mesh	Accelerator	General-purpose
PADDI	Static	FU	Crossbar	Stand-alone	Special-purpose
Pleiades architecture	Static	FU	Hierarchical	Accelerator	Class of applications
XiRisc	Static	FU	Linear	Co-processor	General-purpose
Cognigine-VISC	Static	FU	Other	DSP	General-purpose

Table 1 shows the classification of various reconfigurable processors based on the aforementioned criteria. Following is a brief survey of RP based on the reconfiguration methodology used.

Static and Dynamically reconfigurable processors

Piperench [9] is a pipelined reconfiguration fabric. It has multiple pipes and each pipe consists of a linear array of reconfigurable CLBs. The interconnect structure is linear enabling data transfer from left to right. The amount of computation that can be done by each CLB is small. Hence realization of kernels with large execution time is not possible. Performance is achieved via pipelining. Run-time reconfiguration is used to control data flow between CLBs. Though initial version of Piperench did not support memory access, this has been corrected in Piperench+.

Field Programmable System On a Chip (FIPSOC) [10] consists of a microcontroller, a programmable digital unit, a configurable analog unit and internal memory. The digital unit is known as Digital Macro Cell (DMC) and contains a 2-D array of 4-bit programmable elements. The 4-bit programmable elements contain a sequential block, combinational block and internal routing resources. These elements are programmed with the help of Look-Up Tables (LUTs). The microcontroller has complete access to these LUTs and can write onto them in one clock. This facilitates both static as well as dynamic reconfiguration. Routing is done via 16 horizontal channels per row and 24 vertical channels per column. Switching matrices are used to interconnect horizontal and vertical routes. Special nets are used for clock transfer. Clock distribution network is used for power reduction.

Chameleon Reconfigurable Communications Processor [11] consists of a 32-bit embedded core, 32-bit reconfigurable fabric and a high-speed system bus. The Reconfigurable fabric consists of slices which contains three tiles. Each of these slices can be reconfigured separately. Each tile is made up of 32-bit reconfigurable datapath units, a local storage memory, 16x24 multiplier and a Control unit. There is complete connectivity between Datapath units. Each DPU is connected to its neighbors and other DPUs in the same slice with a delay of one clock and is connected with DPUs in other slices with a delay of 2 clocks. The routabilty is dynamic. Since the configuration information is built into the instructions, reconfiguration time is minimal.

Multiple Alu archiTecture with Reconfigurable Interconnect eXperiment (MATRIX) [12] consists of an array of 8-bit Functional Units (FU) interconnected through a configurable network. Each FU consists of 256x8 memory, control logic and an 8-bit ALU. A three-level hierarchical interconnect is used to connect the Fus. The three levels are Nearest Neighbors, length four bypass and global buses. FU port inputs and network lines can be configured either statically or dynamically.

Configurable System on Chip (CSoC) [13] consists of a 4x4 XPP core from PACT, an ARM processor, memory elements and interconnects. The Xtreme Processing Platform (XPP) core is a hierarchical array of coarse grain Processing Array Elements (PAE). A number of PAEs are clustered to form Processing Array Cluster (PAC) each of which is associated with a configuration manager. This manager has distributed access over the configuration memory of the CAEs. Hence, configuration and processing can be done in parallel. Dataflow is stream-based.

Reconfigurable Pipelined Datapath (RaPiD) [14] is a pipelined reconfigurable unit with multiple pipelines. A single pipeline is designed as a linear array of reconfigurable units. Both static and dynamic control are available. Static control is used to reconfigure the Functional Units as in any typical FPGA. Dynamic control is used to determine dataflow during run-time. Each Functional

Unit can perform multiplication and ALU operations on one word of data. External memory accesses are not handled by the reconfigurable array.

Run-time reconfigurable processors

Garp [15] architecture consists of a single-issue microprocessor interconnected with a reconfigurable array of Configurable Logic Blocks (CLBs). The reconfigurable array is similar to a FPGA, the only difference being in the granularity of the processing elements. The processing elements are connected using a crossbar interconnect. A wide data-path is provided between the memory and the array, thereby increasing the bandwidth. Moreover, preloading of configuration bitstream into configuration memory of the reconfigurable array is possible. The optimizations for a particular application are mainly obtained using a well-defined compiler. Morphosys [16] consists of the following parts: a 8x8 Reconfigurable Cell array with context memory, a TinyRISC processor used as controller, Frame buffer and a DMA controller. RC array is a 8x8 2-D array interconnected using a 2D-mesh architecture along with full row/column interconnectivity. Each RC has an ALU, multiplier and a register file. The context bitstreams are transferred in two modes - row broadcast and column broadcast. Sea of Processors (SOP) [17] consists of a large number of logic elements. Each logic element is connected to 8 neighboring elements. Connectivity is achieved using switching blocks. Switching blocks are of two types: point-to-point and common bus. Logic element consists of Configurable Arithmetic Blocks (CAB) which consist of a full adder and flip flops. Memory elements are designed as functional memory.

Chimaera [18] is a RISC architecture with a Reconfigurable Functional Unit (RFU). The RFU is a coarse-grain FPGA. Plado [19] consists of a simple Motorola core processor along with a Xilnx FPGA as reconfigurable co-processor. This was designed for proof-of-concept. Emphasis was more on compiler techniques than on hardware realization. PRISM [20] is a RAM-based FPGA system. It was designed as a proof-of-concept. There is a strong link between the compiler and the underlying hardware. Common Minimal Processor Architecture with Reconfigurable Extension (COMPARE) [21] is a typical RISC architecture. It has a 3-stage pipeline consisting of Instruction Fetch/Decode unit, Reconfigurable Functional Unit and Load/store unit and a register file. RPU has the common ALU operations along with a number of Configurable Arithmetic Units (CAU). ILP can be exploited as CAU can execute in parallel with ALU. The CAU is made up of 4-bit and 8-bit LUTs. Concise [22] has a RISC architecture similar to CoMPARE. Chess array [23] is a FPGA-type reconfigurable plaform with 4-bit ALUs interconnected using 4-bit data buses. A nearest neighbor switching network is used to connect the various ALUs. The switching boxes can also be used to store data when not in use. Additional connectivity is provided using extra long wires. The local connectivity is made ore dense than global connectivity. Spyder [24] is a reconfigurable superscalar co-processor using FPGAs. Spyder has a fixed underlying VLIW structure with a single control unit and multiple reconfigurable execution units. For REUs, the I/O ports are fixed, but the functionality can be defined by the user.

Reconfigurable Architecture Workstation (RAW) [25] processor is a chip containing a number of identical tile processors. Each tile processor consists of local memory, static switch and a dynamic router. The static network is a mesh architecture, while the dynamic routing network is a partial crossbar architecture. Static switches are used for simple operations like move, load etc, whereas dynamic router is used for data transfer. Each tile consists of 32 KB each of data and instruction memory. Data transfer is done via static switches which are pipelined. KressArray [26] consists of several reconfigurable Data Path Units (rDPUs). Each of the data paths support

multiple data-widths. Communication is done via Nearest Neighbor logic, which is programmable at run-time. KressArray thus resembles a systolic array, the only difference being that the interconnects between each processing element is programmable. Colt Configurable Computing Machine (CCM) [27] consists of 16 Functional Units interconnected using a 4x4 mesh architecture. It also contains a 16x16 multipliers and an additional programmable bus to connect units that need high bandwidth. This architecture supports both bit-level and word-level data transfers which makes it different from conventional FPGAs. Colt CCM is reconfigured using a technique known as Worm-hole run-time reconfiguration. Here multiple streams, each consisting of the configuration as well as computation-data information, are fed into the pool of Functional Units. The header information associated with each stream steers that particular stream through a set of FUs that need to be reconfigured by that stream. This architecture is best suited for high-performance computations.

Reconfigurable Multimedia Array Coprocessor (REMARC) [28] consists of 32-bit processors known as nano-processor. Each nano-processor consists of a program memory. This contains a list of instructions to be executed on that particular processor. But the nano-processor does not have control over the same. Every cycle, a PC value is transmitted to each nano-processor which determines the index of the instruction to be executed. The processors are connected with the help of Horizontal and Vertical bus architectures. Dynamically Reconfigurable hardware Architecture for Mobile systems (DReAM) [29] is a datapath oriented architecture and consists of numerous Reconfigurable Processing Units each controlled by a RPU controller. The interconnect architecture is hierarchical in nature. The control design is also hierarchical. In addition to the nearest neighbor connectivity, there are two global buses. RPU consists of two Reconfigurable Arithmetic Processing (RPU) units, a speading datapath, a communications controller and two dual-port RAMs. Each RAP is built around a 8-bit multiplier and is capable of performing ALU and FSM-type operations. This architecture is best suited for wireless applications.

Static reconfigurable processors

Configurable Algorithm-adaptive Instruction Set Topology (CALISTO) [30] incorporates a adaptive instruction-set architecture into a communications processor. Rabaey et al [31] have developed LP-PGA, an improvement over existing FPGAs. They observed that 65% of the power consumption was due to interconnects. A hybrid interconnect structure consisting of Nearest Neighbor, Mesh and Hierarchical interconnects has been proposed. Configurable Logic Blocks (CLBs) have been realized using 3-input Look-up Tables (LUTs). A combination of 4 such LUTs has been found to be optimal. Programmable Arithmetic Devices for high-speed DSP (PADDI) [32] is a typical coarse-grain reconfigurable architecture. It contains several programmable Execution Units (EXUs) interconnected through a configurable crossbar network. Each EXU contains a 16-bit adder and shifter logic along with registers. This architecture is meant for high-speed realization of data paths.

Pleiades [33] is a heterogeneous reconfigurable DSP platform. It consists of a microprocessor and a group of reconfigurable modules also called satellites. The satellites are functional units which are capable of performing operations like MAC, SAD etc. Nearest Neighbors, Mesh and Hierarchical interconnects are all considered. A cost function is defined to determine which interconnect structure is best-suited for a given application. The computationally intensive kernels are mapped on to the satellites until all the satellites have been assigned. Since the entire

application needs to be mapped onto the architecture, it is not always possible to meet timing constraint. In other words, Pleiades does not provide for hardware virtualization.
Dedicated implementations in the reconfigurable arena for the graphics class of applications, Cohen Sutherland Line Clipping, Mid point Ellipse scan conversion, the bit-Blt (bit block transfer) algorithm and Phong Shading have also been carried out([36], [37] and [38]).
XiRISC [34] is a load/store VLIW processor with a reconfigurable pipeline. This pipeline is realized using Pipelined Configurable Gate Arrays (PiCoGA). This array is tightly coupled with the main processor. There is dedicated control logic to direct control flow in a single pipeline. Programmable interconnects exist between different pipelines as well. XiRISC is a 32-bit architecture. The Cognigine Variable Instruction Set Communication (VISC) [35] architecture is a compile-time configuration based processor. It consists of 16 Reconfigurable Communications Units (RCU). Each RCU has 4 execution units capable of performing 64-bit computations. Amongst these 4, two are meant for ALU operations and two for bit-level manipulations. It has a separate instruction cache to hold instructions. Any given application can be compiled and the corresponding Variable Instruction Set can be obtained. Hence, the emphasis is on the Cognigine C compiler and not on the underlying architecture.
There are certain drawbacks to the approaches taken by contemporary efforts:
The application analyzers are either manual in nature or do not have an efficient automated approach to search for patterns between the Control Data Flow Graphs (CDFGs) of algorithms that span abstract control data flow structures such as nested loops, hammock structures etc, which are intend to be executed on the processor by reconfiguring a previously configured algorithm. The fine granularity based approaches taken by contemporary research efforts have not considered the problem of designing the hardware based on the amount of reconfigurability and scheduling CDFGs such that the amount of on-chip communication for resource and task allocation is minimized. In most cases a fixed hardware is used and reconfigurability is explored based on that hardware. In some cases, researchers have used brute force search methods for exploring the granularity of the processing modules. When CDFGs involving extensive control nodes need to be mapped onto a limited number of resources (processing elements) on the chip, configuration aware scheduling becomes critical.

Current approaches towards building a reconfigurable processor are targeted towards general purpose computing or mapping applications onto off the shelf FPGAs or developing specialized architectures for a class of applications with an intuitive approach towards the design. The increasing demand for power and configuration time aware processing with stringent constraints for flexibility necessitates the design and development of a dynamically fast reconfigurable processor with awareness towards lowering power consumption. The methodology for the design of such a reconfigurable processor is now presented.

3 The Research Problem

The research problem of designing a framework/methodology for the development of fast, dynamically reconfigurable processors consists of 3 primary tasks:

1. To partition the CDFGs of applications into 'reconfigurable' and 'non-reconfigurable' regions. The reconfigurable regions in the CDFGs must then be executed by customized and individually designed monolithic entities (CLUs) on the processor. This partitioning is an important step because, larger amounts of commonality in the tasks of the CDFGs represent lower amount of reconfiguration and hence faster task swapping on the same

units in silicon. This also indirectly reduces the power consumption by lowering the amount of on-chip inter CLU communication.

2. Determination of the number of 'instantiations' of these entities to extract parallelism (instruction, data and group of instruction level) & optimize resource utilization.
3. Scheduling the CDFGs onto CLUs, with awareness for reconfiguration time.

In the following section we provide a conceptual guideline to design this framework.

3.1 Conceptual guideline

The guideline consists of choosing an assortment of computationally intensive algorithms, followed by discrete and distinct steps that lead to the design of the framework.

1. Choosing the algorithms that will constitute a benchmark suite. Target algorithms to be executed on the reconfigurable processor are from the set of media processing and graphics algorithms. These include the algorithms of advanced profiles of MPEG-4 (main, core etc.), and graphics related algorithms such as Line Clipping, Shading, Ellipse scan conversion and block transfer algorithms.
   These operations have complex geometric computations and large control flow structures. These are typically implemented on media accelerator cards or high end computing machines. There is scope for large amounts of parallelism as well as sufficient correlation amongst the applications. Literature survey shows that there have been implementations of some of these algorithms on FPGAs. The high levels of parallelism they offer can be exploited in FPGA forms of programmable architectures.
   These algorithms share the property of having inherent parallelism at both data and task levels. Programmable arrays of spatial processing elements have an inherent ability to exploit the data and task parallelism in applications. But the complexity of the processing elements and their interconnectivity varies from algorithm to algorithm. Each of these algorithms has found implementations in FPGAs, specialized ASICs and general purpose computing machines.
   We propose a tool for this purpose. Our tool will include portions of CDFGs (Control Data Flow Graphs) which will include control / branch instructions (points or nodes). We believe that applications are best analyzed in the form of CDFGs. These belong to the 'graph' class of data structures. Finding isomorphism or near isomorphism for graphs or sub-graphs will produce the set of Largest Common Sub-graphs, that can be implemented as ASICs in a fully customized fashion. This will remove the need to have LUTs on a large scale on the reconfigurable platform. Therefore deviating from the architecture model of programmable "Gate Arrays".
   In order to exploit the concept of reconfigurability + parallel spatial execution, and yet maintain reduced amount of reconfigurations, correlation in the nature of the applications that are aimed to be ported on the processor needs to be exploited. This methodology helps by eliminating switches (Pass transistors) that are always or mostly closed (on) for the same connection on a given route (within the same hierarchy level or across hierarchy levels). Therefore we intend to show that by using the proposed methodology of determining the correlation in processing and communication needs of target algorithms we can optimize the Spatially Programmable Array of Elements (SPAE) class of architectures for a range of applications.
   When CDFGs involving extensive control nodes structures need to be mapped onto a limited number of reconfigurable resources (processing elements) on the chip, task and

resource scheduling with awareness to configuration times and control structure behavior becomes critical. To support scheduling that achieves near optimal solutions without the use of dedicated on-chip dynamic schedulers, a strategy of static scheduling has been chosen. We believe that since a generic set of media oriented algorithms can be ported onto the reconfigurable processor, an extensive set of scheduling situations arising from CDFGs must be theoretically analyzed and solutions be prepared for them. The set of solutions shall be in the form of a collection of algorithms that execute on a Network Schedule Manager (NSM) that resides on the chip. These are intended to modify the schedule table that is stored on the NSM, at run time as and when the need arises.

a) Identify common clusters in these using a set of steps that form CDFGs from optimized gcc compiled assembly files, extract possible parallelism in regions of the application that offer such possibilities and groups portions of the CDFG into reconfigurable regions called zones or clusters through a graph matching algorithm through canonical label comparisons.
   Note: The CDFGs now consist of 2 types of node groups:
   a) Clusterized (These nodes are hence forth referred to as Coarse Grain Processing Elements - CGPE)
   b) Ungrouped or non-clusterized (These nodes are hence forth referred to as Fine Grain Processing Elements - FGPE)

3. All CGPEs will be represented as "Behavioral VHDL" modules. The FGPEs will be represented through LUTs (a group of {2-3} 4-input-LUTs).
   Note: All modules (CGPEs & FGPEs) belonging to an application will be connected through a Rent's Hierarchical Network Architecture [1]. LUTs will be modeled using a combination of "Behavioral" & "Structural" style VHDL.
4. If there are multiple occurrences of a Processing Element within an Application (for example within application (i) Gauss Jordan Elimination),
   - Allocate appropriate number of "Instantiations" based on a partitioning scheme applied on the CDFG.
     Note: If (# of Instantiations < # of Occurrences)
     → Multiple successive configurations (Reconfigurations) of the network for an application is necessary.
5. Static Schedules for each application is obtained. There are 2 behaviors of the CDFG which we will permit to influence the static schedule:
   a. Conditional expression evaluations: We take care of this, by obtaining schedules based on Improved PCP and Branch-&-Bound techniques for each conditional expression. The schedules are merged towards the worst case.
   b. Iterative or loop behavior. In the 2nd pre-processing step we have loop unrolled all iterative segments with the dual properties of: Absence of any conditional branch instructions inside the loop and Known # of iterative count at compile time. Therefore only loops that do not satisfy these 2 properties remain to be considered. For all such cases, we take the approach of assuming the iteration count obtained whenever possible from trace statistics. In cases where trace statistics cannot be obtained a single iteration is considered with the time for that iteration being determined by the critical path in the loop.

6. The Network architecture consists of switch boxes and interconnection wires. The architecture will be based on [1]. This will be modeled as a combination of "Behavioral" & "Structural" style VHDL. Modifications that will be made are:
   a. The Processing Elements derived in step 3 will be used instead of the 4 input LUTs that were used in Andre's model.
   b. RAM style address access will be used to select a module or a switch box on the circuit.
   c. Switch connections that are determined to be fixed for an application will be configured only once (at the start of that application).
   d. Switch connections that are determined to be fixed for all applications will be shorted and the RC model for power consumption for that particular connection will be ignored for power consumption calculations.
   e. The # of hierarchy levels will be determined by the application that has the maximum # of modules, because there is a fixed number of modules that can be connected
7. There will be 1 Network Schedule Manager (NSM) modeled in "Behavioral" & "Structural" style VHDL will store the static schedule table for the currently running application. The NSM collects the evaluated Boolean values of all conditional variables from every module.
8. For placing modules on the network 2 simple criteria are used. These are based on the assumption that the network consists of Groups of 4 Processing Unit Slots (G4PUS) connected in a hierarchical manner.
   Note: A loop could include 0 or more number of CGPEs.
   Therefore the following priority will be used for mapping modules onto the G4Pus:
   a. A collection of 1 to 4 modules which are encompassed inside a loop shall be mapped to a G4PUS.
      i. If there are more than 4 modules inside a loop, then the next batch of 4 modules are mapped to the next (neighboring) G4PUS.
      ii. If # of CGPEs in a loop ≥ 2, then they will have greater priority over any FGPEs in that loop for a slot in the G4PUS.
   b. For all other modules:
      i. CGPE Modules with more than 1 Fan-in from other CGPEs will be mapped into a G4PUS.
      ii. CGPE Modules with more than 1 Fan-in from other FGPEs will be mapped into a G4PUS.
      Note: The priorities are based on the importance for amount of communication between modules. Both Fan-ins and Fan-outs can be considered, for simplicity, we choose Fan-ins to CGPEs only.
9. Time estimation.
   Time to execute an application for a given area (area estimate models of XILINX FPGAs and [1] architectures can be used for only the routing portion of the circuit.) and a given clock frequency can be measured in VHDL.
   The Time taken to swap applications (reconfigure the circuit from implementing one application to another) is dependent on the similarity between the successor and predecessor circuits. We will measure the time to make a swap, in terms of # of bits required for loading a new configuration. Since a RAM style loading of configuration bits will be used, it is proven [2] to be faster than serial loading (used in Xilinx FPGAs). We expect speed up over the RAM style due to 2 reasons:
   a) The address decoder can only access one switch box at a time. So the greater the granularity of the modules, the fewer the number of switches used and hence configured.

b) Compared to peer architectures which have only LUTs or a mixture of LUTs and CPGEs with low granularity (MAC units), we expect to have CGPEs of moderate granularity for abstract control-data flow structures in addition to FGPEs. Since these CPGEs are derived from the target applications, we expect their granularity to be the best possible choice for a reconfigurable purpose. They are modeled in "Behavioral" VHDL and are targeted to be implemented as ASICs. This inherently would lead to a reduced amount of configurations.

The Time taken to execute each application individually will be compared to available estimates obtained for matching area and clock specifications from work carried out by other researchers.

4 The Methodology

This section deals with the detailed description of the 3 primary research problems, the associated literature survey and proposed solutions. It starts with a description of the process of identifying reconfigurable regions in the applications, followed by the technique to determine the number of processing units for high resource utilization and lastly discusses the aspects of scheduling the tasks onto the processing platform..

4.1 Identification of Reconfigurable Clusters

A Control Data Flow Graph consists of both data flow and control flow portions. In compiler terminology, all regions in a code that lie in between branch points are referred to as Basic Blocks. In order to identify regions within a CDFG and those between CDFGs, that have nearly identical control with embedded data flow structures, it is necessary to examine the possibilities of graph homomorphism beyond a basic block. But since, CDFGs are CFGs with embedded DFGs, where a basic block can be interpreted as a DFG, it is first useful to perform a mach between the DFGs constituting a CFG. Code movement can also be performed to result in potential speed ups and identification of larger reconfigurable regions (modified DFGs). But this process can be done in a second pass if there is potential for such speedups and reconfigurations in the applications being considered. The DFG matches are found using the graph matching algorithm described in the following section. Once a potential match is found at the level of a DFG or a modified DFG, groups of such DFGs constituting abstract structures such as a hammock structure, nested loops etc. are searched for matches. A successful match at this level would indicate homomorphism at both the data and control flow levels of the graph. Since matches are investigated at levels of basic blocks, potentially modified basic blocks, and abstract control data flow structures, regions which are not simple basic blocks, for a generic purpose shall be denoted uniformly as zones. Details on the algorithms used to extract zones and identify control structures to form CFGs from an assembly file are given in appendix A. We have used the industry standard gcc compiler with the following optimizations set (redundant expression elimination, constant propagation, copy propagation, constant folding, dead code elimination, scalarization, local register allocation, global register allocation, register targeting, call in-lining, code hoisting and sinking).

4.1.1 Graph Matching

We will fist provide a brief introduction to the area of graph matching and then provide a literature survey of existing techniques. Then the proposed techniques will be discussed with the merits and demerits.

4.1.1.1 Basic Definitions and Terminology

A graph G = (V,E) in its basic form is composed of vertices and edges. V is the set of vertices (also called nodes or points) and $E \subset V \times V$ is the set of edges (also known as arcs or lines) of graph G. The order (or size) of a graph G is defined as the number of vertices of G and it is represented as |V | and the number of edges as |E|.

If two vertices in G, say $u, v \in V$, are connected by an edge $e \in E$, this is denoted by e = (u, v) and the two vertices are said to be adjacent or neighbors. Edges are said to be undirected when they have no direction, and a graph G containing only such types of edges is called undirected. When all edges have directions and therefore (u, v) and (v, u) can be distinguished, the graph is said to be directed.

In addition, a directed graph G = (V,E) is called complete when there is always an edge $(u,u') \in E = V \times V$ between any two vertices (u,u') in the graph.

Exact and inexact graph matching

The graph matching problem can be stated as follows:

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, with $|V_1| = |V_2|$, the problem is to find a one-to-one mapping $f : V_1 \rightarrow V_2$ such that $(u,v) \in E_1 iff (f(u), f(v) \in E_2$. When such a mapping f exists, this is called an isomorphism, and G_1 is said to be isomorphic to G_2. This type of problem is considered to be exact graph matching.

When an exact match cannot be found between two graphs, (for instance if the number of vertices are different), then finding the best matching between them is called homomorphism. In this case, the matching aims at finding a non-bijective correspondence between a data graph and a model graph. In a homomorphic graph matching problem, if we assume $|V_1| < |V_2|$, the goal is to find a mapping $f' : V_2 \rightarrow V_1$ such that $(u,v) \in E_2 iff (f(u), f(v) \in E_1$. This corresponds to the search for a small graph within a big one. An important sub-type of these problems are sub-graph matching problems, in which we have two graphs G = (V,E) and G' = (V',E'), where $V' \subseteq V$ and $E' \subseteq E$, and in this case the aim if to find a mapping $f' : V' \rightarrow V$ such that $(u,v) \in E' iff (f(u), f(v) \in E$. When such a mapping exists, this is called a subgraph matching or subgraph isomorphism.

4.1.1.2 Literature Survey

Exact graph matching: graph isomorphism

This category of graph matching problems has not yet been classified within a particular type of complexity such as P or NP-complete. Some papers in the literature have tried to prove its NP-completeness when the two graphs to be matched are of particular types or satisfy some particular constraints [39, 40], but it still remains to be proved that the complexity of the whole type remains within NP-completeness. On the other hand, for some types of graphs the complexity of the graph isomorphism problem has been proved to be of polynomial type. An example is the graph isomorphism of planar graphs, which has been proven [41] to be of polynomial complexity.

Exact sub-graph matching or sub-graph isomorphism:

This particular type of graph matching problems has been proven to be NP-complete [40]. However, some specific types of graphs can also have a lower complexity. For instance, the particular case in which the big graph is a forest and the small one to be matched is a tree has been shown to be of a polynomial complexity [40, 42].

Inexact graph matching: graph and sub-graph homomorphisms:

In inexact graph matching, where we have $|V_1| < |V_2|$, the complexity is proved in [43] to be NP-complete. Similarly, the complexity of the inexact sub-graph problem is equivalent in complexity to the largest common sub-graph problem, which is known to be also NP-complete.

Note: The problem of trying to find clusters in CDFGs lies in this category.

Some graph matching problems are based on the idea of having more than one model, and on performing graph matching to a database of models so that the model that best approaches the characteristics of the data graph is selected. Therefore, the aim here is to recognize a model rather than going deeply to recognize each of the segments of the data image. An illustrative example is found in [44] in which decision trees are used for graph and subgraph isomorphism detection in order to match a graph to the best of a dictionary of graphs. [45] proposed the method of node growing for graph searches in databases. This involves the formation of a pool of 2 node graphs. The candidates for the nodes in this pool are all possible nodes from the database of graphs to be searched. Each of these small graphs are compared with every candidate graph in the database. The bottom x % of the matches are then pruned. The remaining 2 node graphs are now gown into 3 node graphs. Every possibility is grown. Then the process of comparison is repeated and pruning carried out. The match between 2 graphs is done through comparison of the canonical labels of their adjacency matrices. Various canonification functions can be used to derive the labels. This method uses the longest possible label. But the drawback of this approach is the lack of weights to the edges. Even without weights complex partitioning schemes are applied to the adjacency matrices to obtain labels for comparison. Others who have worked on template matching based clustering include [46, 47, 48, 49].

Some others have reduced the problem of the largest common subgraph problem into combinatorial optimization problem. For example in the Graduated Assignment method [50] a match between 2 graphs is obtained by formulating the differences between the weighted graphs as an objective function. The authors then try to minimize this objective function.

$$E_{ws}(M) = -\frac{1}{2}\sum_{a=1}^{A}\sum_{i=1}^{I}\sum_{b=1}^{A}\sum_{j=1}^{I} M_{ai}M_{bj}C_{aibj}$$

$$\text{subject to } \forall a \sum_{i=1}^{I} M_{ai} \le 1,\ \forall i \sum_{a=1}^{A} M_{ai} \le 1,\ \forall ai\ M_{ai} \in \{0,1\}$$

Here the M_{ai} and M_{bj} are the same matrix (called matching matrix). The C term is the difference in the weights of 2 edges being compared. The summation basically is a combination of every possible edge comparison between the 2 graphs. So, if there is an exact match between 2 edges then the product of the Ms and C will be 1, else it will be 0. Hence the minimum value of the summation represents the maximum number of edge matches between the 2 graphs. Since a node in a graph can only match up with one node in the other graph, the match matrix should be a permutation matrix. In classical combinatorial

problems, assignment problem corresponds to finding a permutation matrix for a given sample matrix such that the summation of the chosen elements (an element in the sample matrix is chosen if its corresponding entry in the permutation matrix is 1) is maximum. The authors try to convert the given minimization problem into a maximization problem, by expanding the objective function using taylor's series. They then convert the the discrete problem into a continuous version by using a control parameter. This is done by producing an initial match matrix, obtained by exponentiating the error function. This is then subjected to iterative row and column normalization which should result in a doubly stochastic matrix (Sinkhorn's rule). Sinkhorn's rule states that, any positive matrix when iteratively normalized along rows and then along columns will converge into a doubly stochastic matrix. A doubly stochastic matrix is one whose summation along any row or column is positive and less than or equal to one. The newly obtained matrix is again reused in the objective function with a newly increased parameter value. The complexity of this approach is O(lm) where l and m are the number of edges in the 2 candidate graphs.

A review on general purpose probabilistic graph matching can be found in [51], where different types of probabilistic graphs, different techniques for their manipulation, and fitness functions appropriated to use for these problems are presented. Some have also used Fuzzy set theory as a means to create vertex and edge attributes to be applied to graph matching. References in the literature using this type of attributes for inexact graph matching include [52, 53]. [54] transforms the graph matching problem into the maximum clique problem and proposes a generic solution free from application domains. [55] proposes the use of a Lagrangian relaxation network to match graphs.

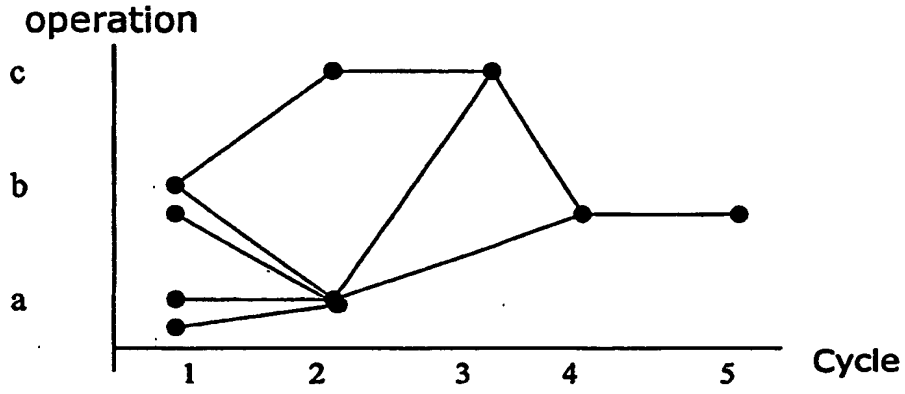
Most of these approaches suffer from some disadvantages. Methods such as the graduated assignment method being iterative, gives no indication as to how fast the objective function converges. It is also an approximation approach since it involves maximization of an error term. The growing nodes method is very slow and cumbersome. Therefore the following method is proposed, which has low complexity and designed specifically for matching conditional data flow graphs.

In the following approaches, we start with CDFGs representing the entire application and which have been subjected to zone identification, parallelization and loop unrolling. The zones / Control Points Embedded Zones (CPEZ) that can be suitable candidates for reconfiguration will be tested for configurable components through the following approaches. *Note: Each Zone / CPEZ will be represented as a graph.*

4.1.1.3 Proposed Approach

All computations are best analyzed for hardware synthesis by representing the applications or algorithms to be implemented, in terms of Control Data Flow Graphs. In the context of CDFGs, there are two primary restrictions on the graphs. The first is that, apart from loops, no other edge can traverse the graph in a direction against the cycle increment. For example, in the graph shown in Figure 10, all edges are directed and flow is from left to right. The other restriction on the graphs is that all CDFGs must be developed based on 2 axes: The x-axis, representing the cycle info, and the y-axis representing the operation or task being performed. By taking advantage of these restrictions, if we can represent graphs in terms of strings, then low complexity algorithms for the purpose of string matching can be effectively used. It is well known that graphs can be represented through adjacency matrices, whose canonical labels can be effectively used as strings. We now have to develop strings from these graphs, the CDFGs. We do that by populating adjacency

matrices such as the one shown in the table of Figure 1. This has an array of indices associated with row indexes, whose rows have one or more entries. An entry (or count in the table) indicates that an edge exists between the source vertex (indicated by a row index) and the destination vertex (indicated by the destination vertex). The array associated with a row index, are the columns in that row where a populated count exists.



		a2	b2	c2	a3	.....
[1,...]	a1	2				
[1, 3,..]	b1	2		1		
	c1					
	a2					
	.....					

Figure 1: A Single Graph with the Adjacency Matrix

Since all zones in the CDFGs need to be represented in this manner, preparing the adjacency matrices cannot be avoided. But while reading out the matrices into canonical labels, all elements in the tables need not be checked for Null value. Therefore a string of tasks, associated with cycle information is obtained as shown below:
a1-a2, a1-a2, b1-a2, b1-a2, b1-c2, a2-b4, a2-c3, c2-c3, c3-b4, b4-b5.

The string of edges (elements) obtained this way will now be sorted using an efficient algorithm such as Merge Sort whose complexity is O(nlogn). Prior to sorting, all cycle info is hidden. It must be noted that since multiple nodes can exist at a particular cycle, a mechanism is required to distinguish between cases of multiple fan-ins or fan-outs to a node and multiple nodes of identical type with lower number of fan-ins or fan-outs. For example, in the figure shown below (Figure 2), the two cases should not be encoded identically. Therefore to distinguish between the two cases, a cycle is split into as many number of

stages as dictated by the maximum node copies of a particular type. In the example shown on the right, node b (op2) in cycle 4 has 3 copies, so cycle 4 is split into 3 stages. The nodes are then spread out among these cycles. The cycles are then renumbered increasingly.

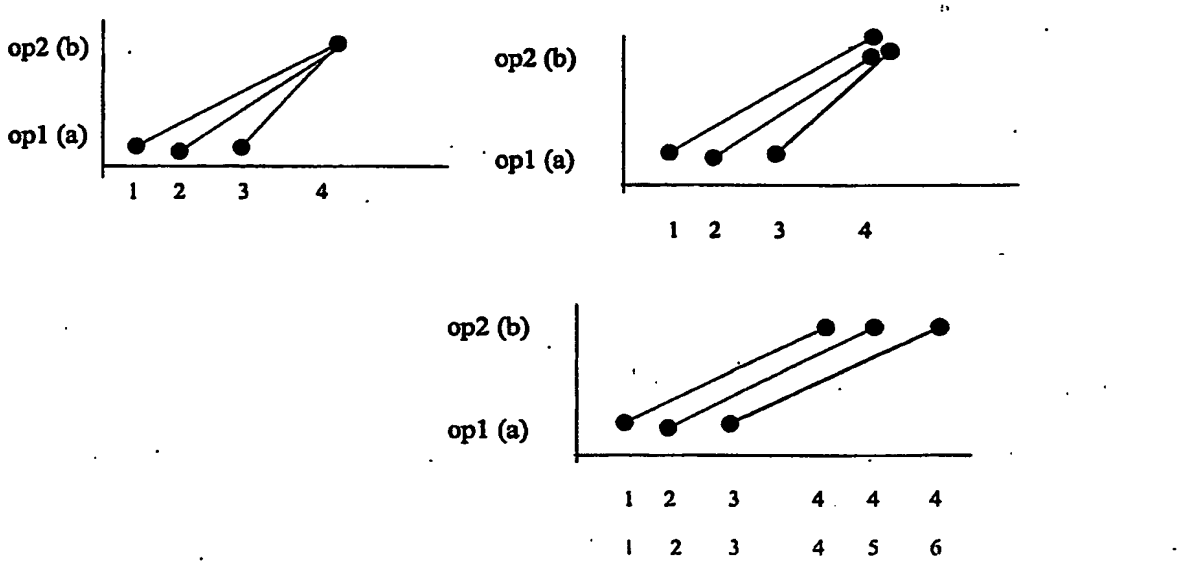


Figure 2: Cycle Splitting and renumbering

This helps in uniquely identifying the labels for the edges and nodes. The cycle information of the nodes is used while identifying a match between edges in similar bin pairs as explained below.
After hiding the cycle information, this process will result in the following string sequence being generated:
aa, aa, ac, ba, ba, bb, bc, cb, cc

The criteria for sorting is based on the fact that an edge consists of 4 basic elements (Source Clock, Source Operation, Destination Clock, Destination Operation). If the Clock information is now hidden, then the SO of two edges are compared and the one with the lower rank is placed to the left. In the example shown, source operation 'a' has a lower rank than 'b' and 'c'. If the SO of the edges are the same, then their DO are compared. The same rule applies: the DO with the lower rank, is placed to the left. In this manner, the string is sorted. Now these pairs of alphabets will be placed into bins. In order to place them the first or the left most pair (aa in our example) is assumed to be the head of the queue. It is placed in the first bin. Then all the following elements in the queue are compared with the head, till a mismatch is obtained. If a match occurs then, that pair is placed in the same bin as the head. Now the first mismatched pair is designated as new head of the queue. This is now placed in a new bin and the process is followed till all elements are in a set of bins as shown in the following Figure (3).

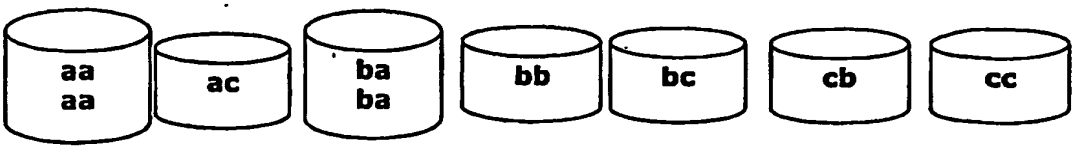


Figure 3: First Graph's edges arranged into a Bin Sequence

The next step is to perform a similar but not exactly the same process for the graph that needs to be compared with the candidate graph, graph # 1.
Consider a second graph, graph #2 as shown in Figure 4.

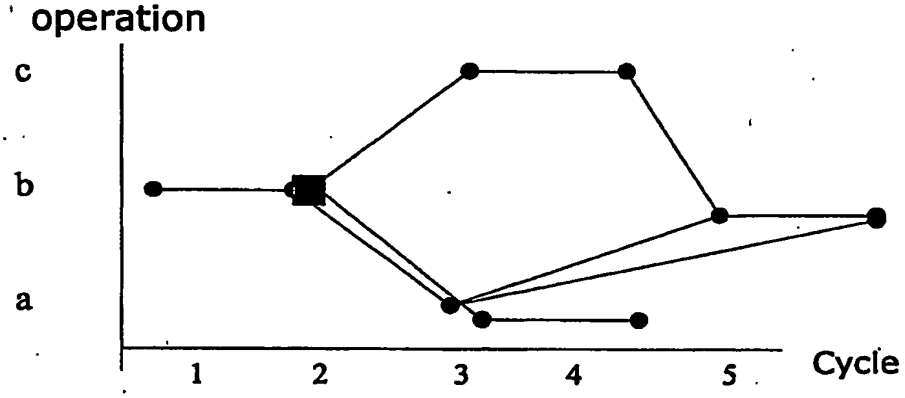


Figure 4: The Second Graph

This graph is converted to a string format in the same manner as graph #1 and this string, as shown below needs to be placed into a new set of bins.

aa, ab, ab, ba, ba, bb, bb, bc, cb, cc

This is done by assigning the leftmost element in the queue to be the head. It is first compared to the element type in the first bin of the old set (aa) [This is termed as the reference bin]. If it checks to be the same, then the first bin of the new set is created and all elements upto the first mismatch are placed in this bin. Then the reference bin is termed as checked. Now the new head type is compared to the first unchecked bin of the reference set. If there is a mismatch, then the comparison is done with the next unchecked bin and so on, until the SO of the element type is different from the SO of the element type in the reference bin. At this point, a comparison of all successive element pairs in the current queue are compared with the head, till a mismatch is met. Then the matched elements are eliminated.
But incase, a match is found between the head of queue and a reference bin, then a new bin in the current set is created and suitably populated. The corresponding reference bin is checked and all previously / predecessor unchecked reference set bins are eliminated.
By this approach, we are eliminating comparison between unnecessary edges in the graphs.
Now a new set of bins for graph 2 is obtained as shown (Figure 5):

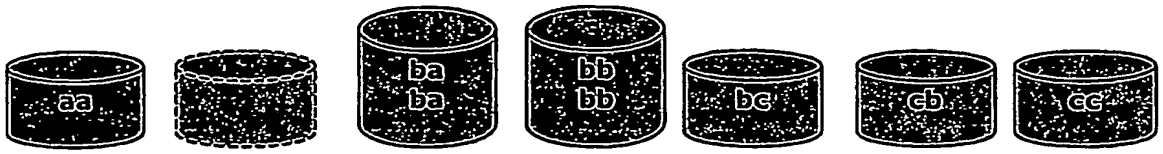


Figure 5: The Second Graph arranged in a Modified Bin Sequence

In the first bin set, the one containing 'ac' is eliminated.
To find the largest Common Sub-graph, we need to operate on corresponding pairs of bins (bins with same SODO) from both queues, where both bins have elements in them. While comparing edges in a bin pair, the cycle information is now used. To establish a match between source(s) to destination node structures, we consider all fan-ins to a destination node. That is, if there exists a match between an edge in one bin and an edge in the corresponding bin pair, then all other fan-ins to the first edge's destination node are compared to all fan-ins to the second edge's destination node. Only if a match is found in all the fan-ins, then the two structures are classified as a match. Then these edges are cancelled out of the bin sequence. Thus all edges are exhausted form the sequence. The set of matched edges now constitute the largest common sub-graph (lcsg). While performing a match all properties of edges & nodes have to be considered: cycle info, type of data transaction (fixed, floating point), bit precision / data width (1,2..4, 8 bits etc..). Once a lcsg is extracted from a pair of basic blocks the lcsg is identified henceforth as a node with a unique label, then matches are searched for in the governing control structure of these basic blocks. This leads to the search for abstract data flow embedded control structures. To perform the match among a group of basic blocks interconnected through control flow edges, appropriate labels are given to lcsgs obtained from basic blocks, modified basic blocks and zones.
The advantage of this approach is that a fan-in structure (multiple source nodes feeding a destination node) is subject to a match from a potential edge match from a bin pair. At the first mismatch in a primary fain-in structure's edge, any further search for matching edges in the secondary structure is abandoned. The search resumes with next edge in the secondary bin pair. If no matching structure is found, then all edges corresponding to the first structure are eliminated from the primary bin sequence. This approach takes advantages of the nature of graphs that represent instruction sequences for computational problems. Analyzing instruction sequences with simpler data structures such as doubly linked lists makes deriving common and configurable clusters very difficult. It is also not possible to extract such common sub-graphs from template matching methods because, that rely heavily on the initial templates and growing of templates beyond 8-10 vertices makes the problem quite complex.

Therefore after subjecting the zones / CPEZs to the cluster (Largest Common Sub-graph) extraction process, the CDFG representing the application as a whole will consist of 2 types of entities: Clusters & non-clusterized zones (or CPEZs). For the purpose of scheduling, they will be termed as 'Processes'. For the purpose of implementation in VHDL, we will model the clusterized processes in 'Behavioral' form. The non-clusterized zones will be implemented on generic LUTs, modeled in 'Behavioral & Structural' style VHDL. An example of such a CDFG is shown in Figure 6, where processes belonging to cluster types 1

& 2 will be mapped onto processing units specialized to execute them, where as non clusterized processes indicated in blue, will be mapped onto generic LUT structures.

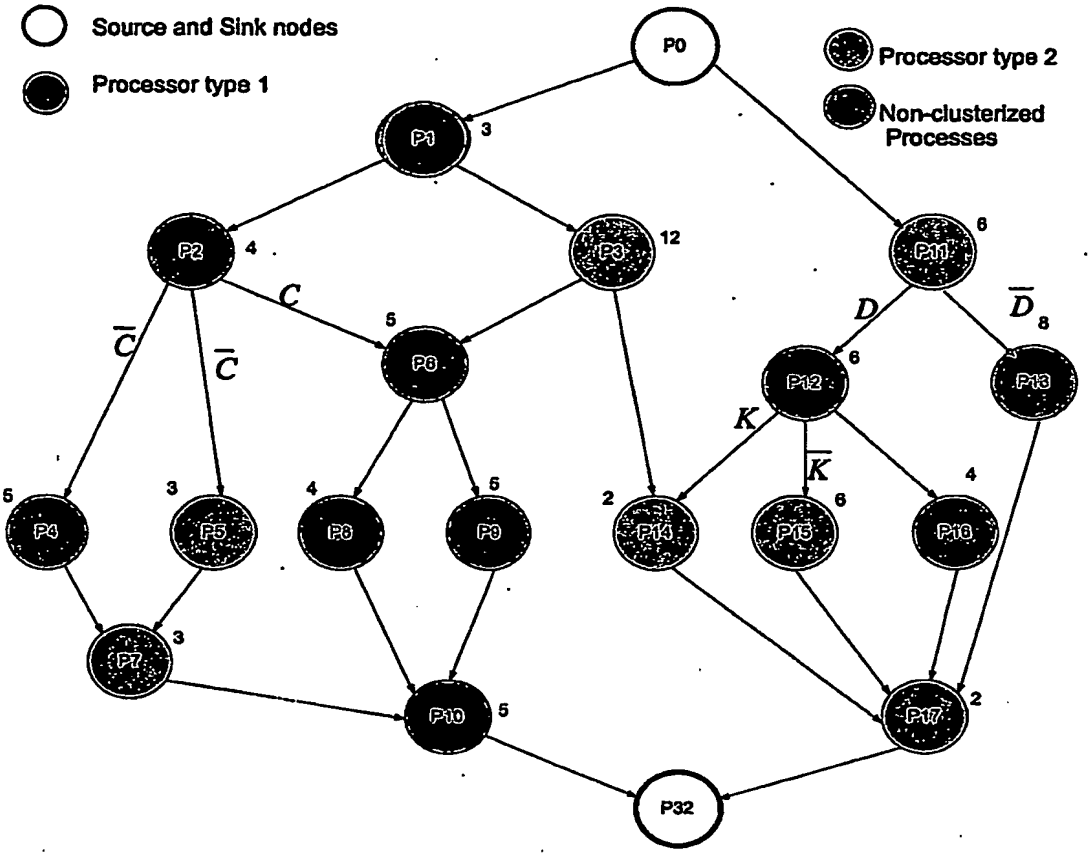


Figure 6: An Example CDFG

4.2 Number of Processing units for a Cluster Type

Once the clusters have been identified, it is necessary to choose judiciously the number of processing units necessary to execute jobs (clusters) belonging to the same type. This process determines the amount of spatial parallelism that the reconfigurable processor can offer. To maximize the utilization of each processing unit, the following divide and conquer approach is followed.

4.2.1 Partitioning by Divide & Conquer Approach

Let the number of resources allocated for Process of type PEi be Ni. In Figure 6, the following configuration has been assumed. There are three processors PE1, PE2 and PE3. N1 = N2 = N3 = 1, one for each type of process.

Determination of number of resources for each type of process:

Obtain the sub-graphs for every possible path. For example, the graph in Figure 6 is used to obtain the path for DCK (sub-graph shown in Figure 7).

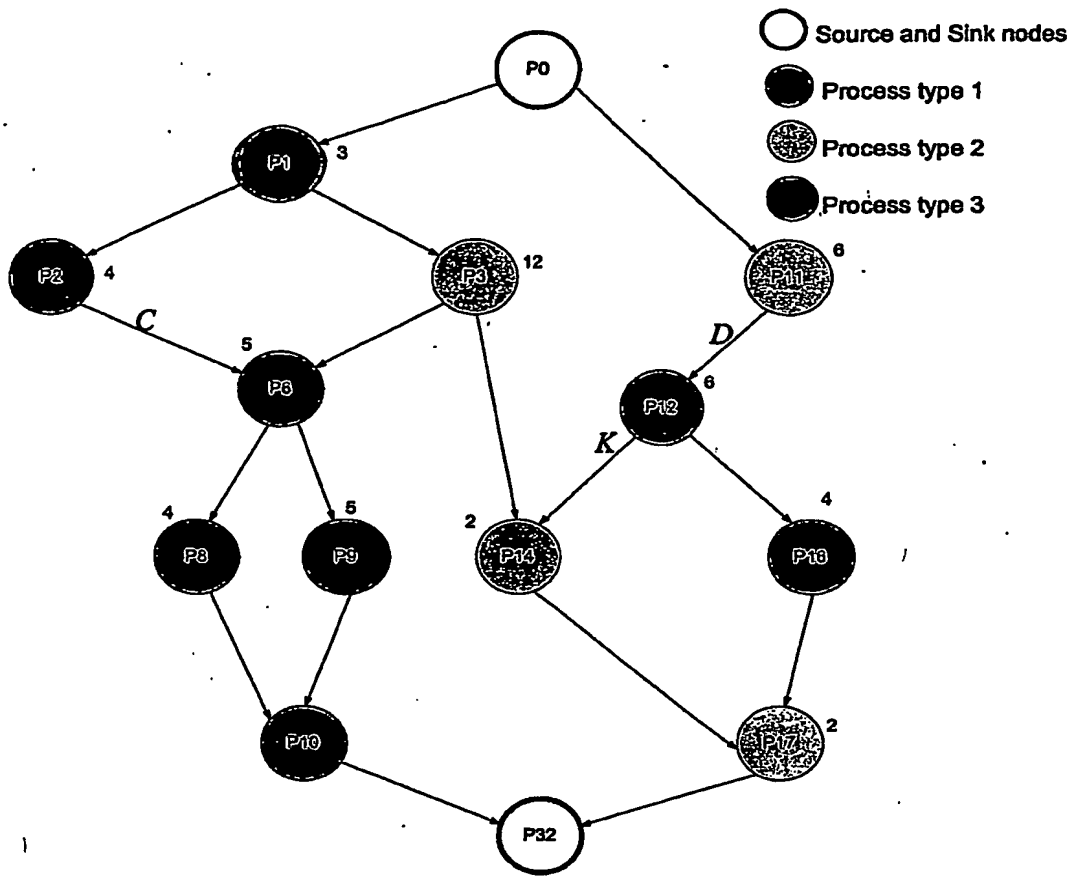


Figure 7: Sub-graph for condition DCK

For this particular example graph, 6 such sub-graphs are obtained, one for each of the conditions: $DCK, DC\overline{K}, D\overline{C}K, D\overline{C}\overline{K}, \overline{D}C, \overline{D}\overline{C}$.

For any sub-graph, we can determine the number of units of each type of processor. This is done by isolating the nodes corresponding to a processor type. For example, in the DCK example taken above, 3 more graphs can be obtained as shown in Figure 8 and 9. It might not make sense to apply this policy to all the 6 sub-graphs. Therefore only those deemed as most likely to be taken should be considered. In case an unlikely path does end up being taken, the clock speed for the general purpose computing resources (the programmable LUTs) must be designed suitably if real-time requirements exist.

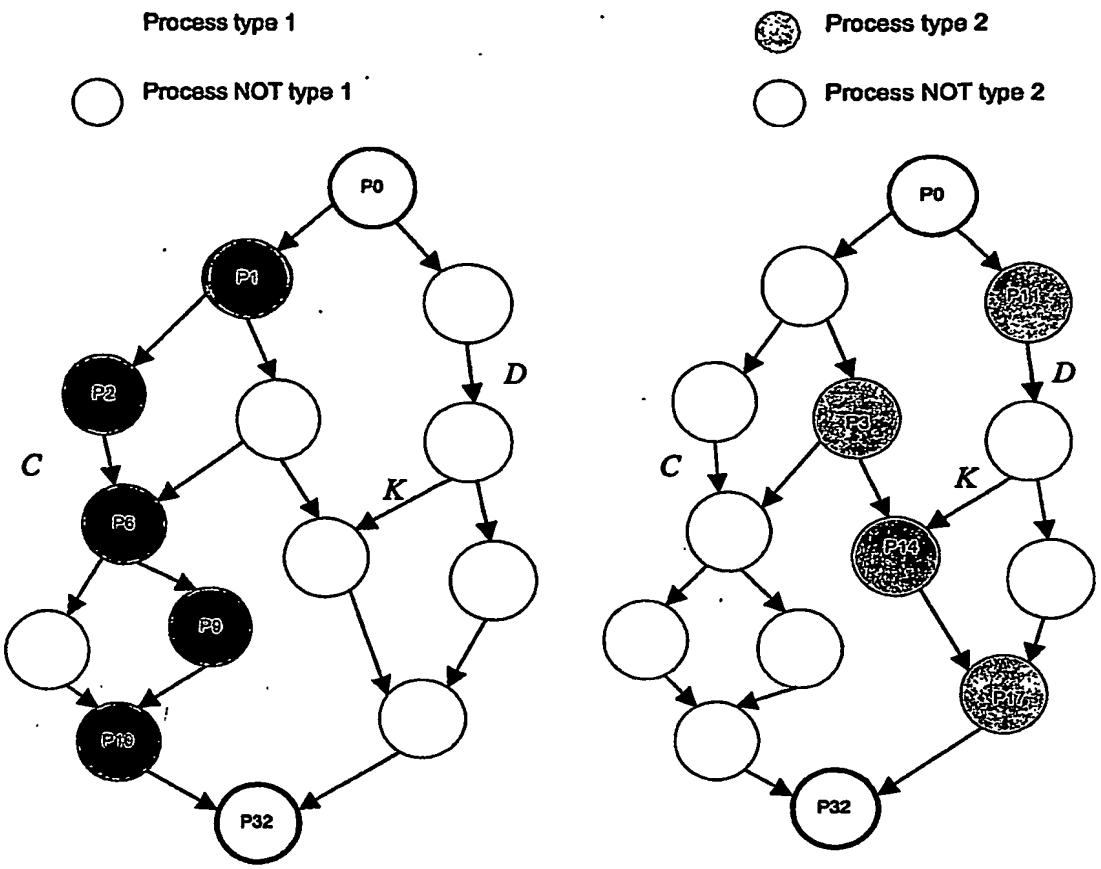


Figure 8: Graphs obtained for Individual Process types 1 and 2

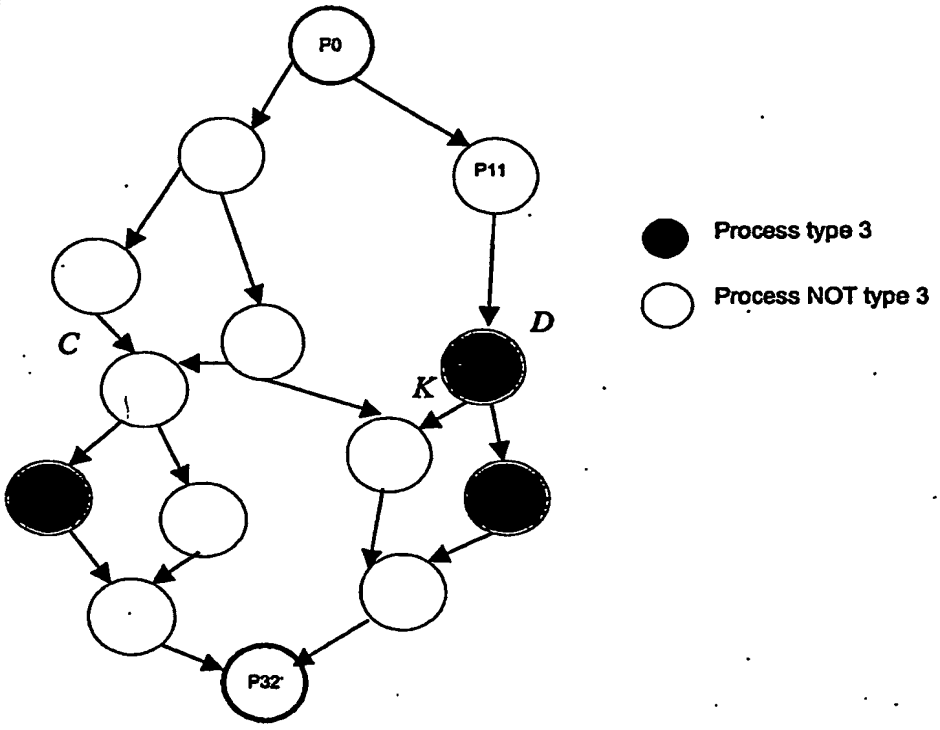


Figure 9: Graphs obtained for Individual Process type 3

For a graph with only a specific process type highlighted, we obtain the number of processor units, by identifying the critical path and separating the graph into processes in the critical path and those outside it. For example if we had a graph as shown in Figure 10, with the critical path marked in dotted line arrows, we would group processes P1, P5, P8, P9, P10, P12 and P13 into the primary group. And we would place all the other processes into another group called the secondary group. If the combined execution time in the primary group is say Tp and the combined execution time in the secondary group is Ts, then we check for the ratio of Tp : Ts. If the ratios are close to 1:1, then it means that most likely, maximum benefit can be obtained by scheduling each of the groups onto 2 parallel processors. If the ratio is 1:x where $x \geq 1$, then in the secondary group, a critical path is identified. Thus the secondary group is similarly split into 2 groups. We proceed in this divide and conquer method till a 1:1:1... or a close ratio is obtained. But if Tp : Ts is x : 1, then there would be an underutilization of resources if additional processing units are allocated. In this case we might be better off using a single resource allocation.

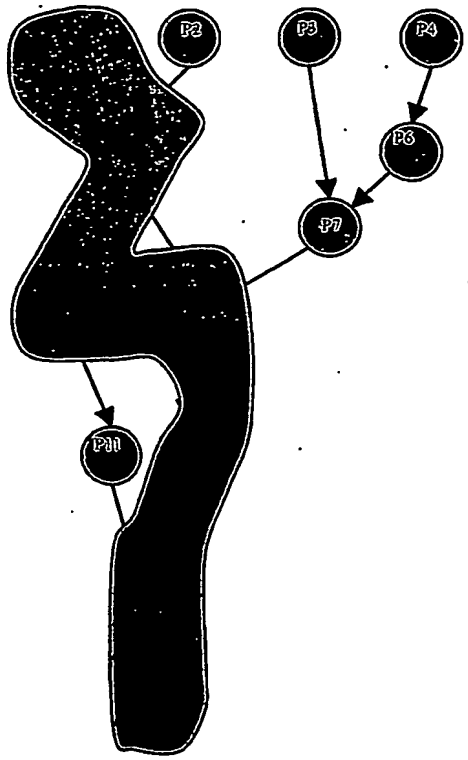


Figure 10: Divide and Conquer Method of Determining Number of Parallel Units

4.3 Scheduling

Once the number of processing units has been chosen, the CDFGs have to be mapped onto these units. This involves scheduling, i.e allocating of tasks to the processing units in order to complete execution of all possible paths in the graphs with the least wastage of resources but avoid conflicts due to data and resource dependencies. In this section we first present a literature survey followed by the proposed scheduling strategy.

4.3.1 Literature survey

We now address the issue of task scheduling. In the graph matching problem, we can include branch operations to reduce the number of graphs. This can be done, if one of the paths of a branch operation leads to a very large graph compared to the other path, or is a subset of the other path. This still leaves us with the problem of conditional task scheduling with loops involved. Since scheduling is applicable to many diverse areas of research, in this section we will not discuss all the work done in scheduling. Instead we focus on those that are relevant to mapping data flow graphs on processors and propose a method most suitable for the

purpose of reconfiguration and compare it with the contemporary methods. Several researchers have addressed task scheduling and one group has also addressed loop scheduling with conditional tasks [57]. A detailed survey of data and control dominated scheduling approaches can be found in [58], [59] and [60]. Chekuri's [61] paper discusses profile driven scheduling based on the earliest branch node retirement scheme. This is applicable for trees and s-graphs. An s-graph is a graph where only one path has weighted nodes. In this case, it is a collection of DAGs representing basic blocks which all end in branch nodes, and the options at the branch nodes are: exit form the whole graph or exit to another branch node as shown in Figure 11. The two DAGs are in blue and violet and the branch nodes are in red with probabilities of exiting the system being p and q. Such a graph is scheduled on a generic set of processing elements such that the branch node with the least schedule length to sum of exit probabilities from the system (upto that branch node) is scheduled as early as possible. The denominator implies how many sub-graphs 'can be completed'. The numerator implies 'how fast this can be done' or 'what is the least amount of time it takes to do it'. Therefore, a low rank indicates that a large number of sub-graphs can be completed in a small amount of time. This is a form of list based scheduling where they try to minimize the expected time of completion.

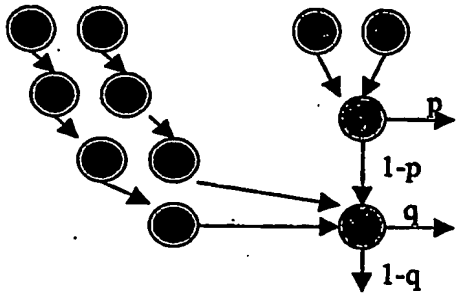


Figure 11: Early Retirement Schedule

The problem with this approach is that it is applicable only to 'small, as defined by the authors' graphs and also restricted to S-graphs and trees. It also does not consider nodes mapped to specific processing elements. Others such as [62] and [63] also consider trace information and string together many basic blocks. Approaches taken by [64] provide hardware support for branch predictions. Among those who have worked on scheduling for conditional flow graphs include [65] and [66,67]. Jha's paper [57] addresses scheduling of loops with conditional paths inside them. This is a good approach as it exploits parallelism to a large extent and uses loop unrolling. But the drawback is that the control mechanism for having knowledge of 'which iteration's data is being processed by which resource' is very complicated. This is useful for one or two levels of loop unrolling. It is quite useful where the processing units can afford to communicate quite often with each other and the scheduler. But in our case, the network occupies about 70% of the chip area [1] and hence cannot afford to communicate with each other too often. Moreover the granularity level of operation between processing elements is beyond a basic block level and hence this method is not practical. And within a processing element, since the reconfiguration distance (edit distance) is more important, fine scale scheduling is compromised because the benefits with the use of

very fine grain processing units is lost due to high configuration load time. [68] paper discusses a 'path based edge activation' scheme. This basically means, if for a group of nodes (which must be scheduled onto the same processing unit and whose schedules are affected by branch paths occurring at a later stage) we know ahead of time the branch controlling values, then we can at run time prepare all possible optimized list schedules for every possible set of branch controller values. In the following simple example shown in Figure 12, the nodes in gray need to be scheduled on the same processing unit. The branch controlling variable is b which can take values of 0 or 1. In case it takes a 0, the branch path in red is taken, else the path in green is taken. In the case where we can know at run time, yet ahead of time of occurrence of the branch paths, the value of 'b', we can prepare schedules for the 3 grey nodes and launch either one, the moment b's value is known.

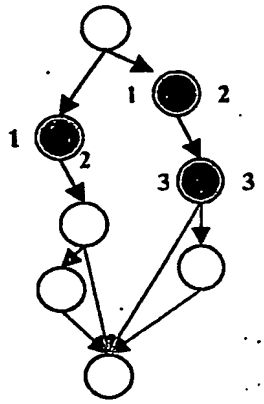


Figure 12: Path based edge activation

This method is very similar to the partial critical path based method proposed by [69]. It involves the use of a hardware scheduler and is quite well suited for our application. But we need to add another constraint to the scheduling: the amount of reconfiguration or the edit distance. In [69] the authors tackles control task scheduling in 2 ways. The first is partial critical path based scheduling, which is discussed above. Although they do not assume that the value of the conditional controller is known prior to the evaluation of the branch operation. They also propose the use of a branch and bound technique for finding a schedule for every possible branch outcome. This is quite exhaustive, but it provides an optimal schedule. Once all possible schedules have been obtained, the schedules are merged. The advantages are that it is optimal, but its has the drawback of being quite complex. It also does not consider loop structures. Other papers that discuss scheduling onto multiprocessor systems include [70], [71] and [72]. Among other works carried out on static scheduling by ([73] and [74]) involve linearization of the data flow graphs. Some others have also taken fuzzy approaches [75] and [76].

4.3.2 Proposed approach

Given a control-data flow graph, we need to arrive at an optimal schedule. Section 4.3.2.1 explains the CDFG. This is followed by the methodology to obtain near optimal schedules. This involves a brief discussion on the PCP scheduling strategy followed by an enhancement to this approach to arrive at a more optimal schedule. Section 4.3.2.3 explains the need to involve reconfiguration time as additional edges in the CDFG. Section 4.3.2.4 talks about ways to handle loops embedded with mutually exclusive paths and loops with unknown execution cycles.

4.3.2.1 Control-Data Flow Graph

A directed cyclic graph has been used to model the entire application. It is a polar graph with both source and sink nodes. The graph can be denoted by G (V, E). V is the list of all processes that need to be scheduled. E is the list of all possible interactions between the processes. The processes can be of three types: Data, communication and reconfiguration. The edges can be of three types: unconditional, conditional. Here a simple example with no loops has been shown in Figure 13.

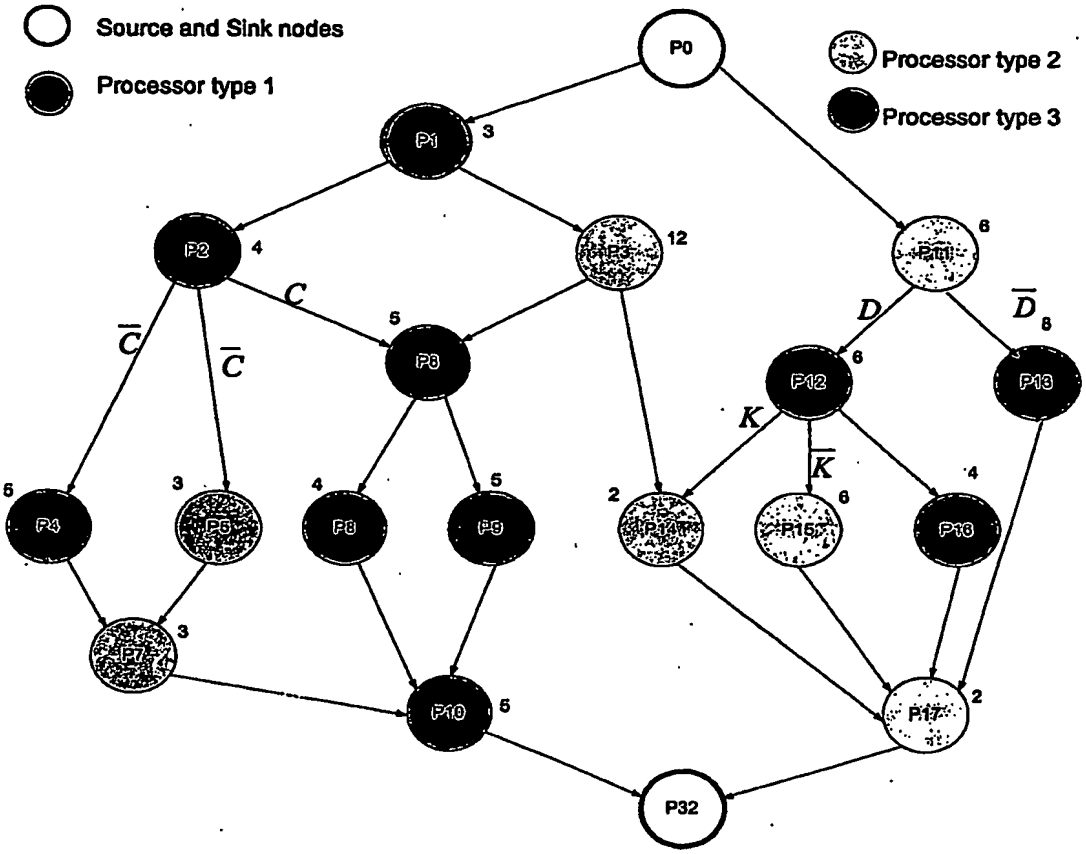


Figure 13: An Example of a Control Data Flow Graph

In the above graph, each of the circles represents a process. Sufficient resources are assumed for communication purposes. All the processes have an execution time associated with them, which has been shown alongside each circle. If any process is a control-based process, then the various values to which the condition evaluates to are shown on the edges emanating from that process circle.

4.3.2.2 Methodology

i. Use PCP scheduling to determine the delays for each possible path of the CDFG and arrange the list of paths in descending order of the delays.
ii. Perform branch and bound based scheduling (which need not be done for every path to reduce the complexity).
iii. Once the final list of all schedules is ready, merge all the schedules by respecting data and resource dependencies.

PCP scheduling:

PCP is a modified list-based scheduling algorithm. The basic concept in a Partial critical path based scheduling algorithm is that if we have a situation as shown in Figure 14 below, where Processes P_A, P_B, P_X, P_Y are all to be mapped onto the same resource say Processor Type 1. P_A and P_B are in the ready list and a decision needs to be taken as to which will be scheduled first. λ_A and λ_B are times of execution for processes in the paths of P_A and P_B respectively, but which are not allocated on the Processors of type 1 and also do not share the same type of resource.

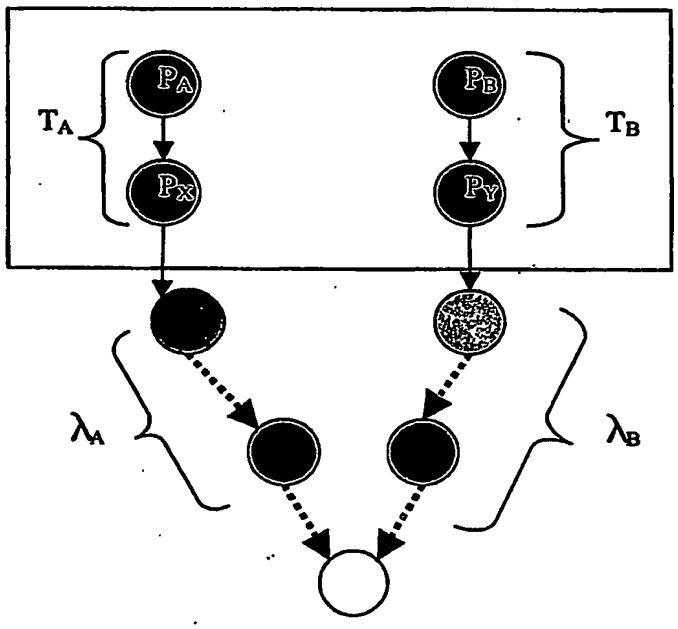


Figure 14: PCP based Scheduling

If P_A is assigned first, then the longest time of execution is decided by the

Max ($T_A + \lambda_A$, $T_A + T_B + \lambda_B$)

If P_B is assigned first, then the longest time of execution is decided by the

Max ($T_B + \lambda_B$, $T_B + T_A + \lambda_A$)

The best schedule is the minimum of the two quantities. This is called the partial critical path method because it focuses on the path time of the processes beyond those in the ready list. Therefore if λ_A is larger than λ_B, a better schedule is obtained if Process A is scheduled first. But this does not consider the resource sharing possibility between the processes in the path beyond those in the ready list. A simple example (Figure 15) shows that if $T_A = 3$, $T_B = 2$, $\lambda_A = 7$, $\lambda_B = 5$, where in processes in the λ_A and λ_B sections share the same resource, say Processor type 2, then scheduling Process A first gives a time of 15 and scheduling B first gives a time of 14. But both the critical path and PCP as proposed by Pop suggest scheduling A first.

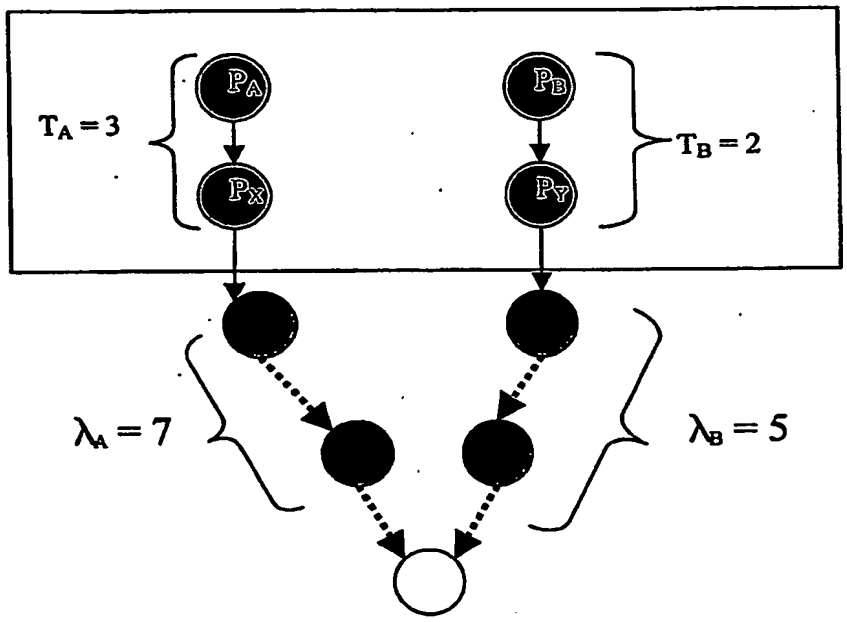


Figure 15: PCP Scheduling with Resource Dependencies in the Partial Path Region

The difference is because, if the resource constraint of the post ready list processes is considered, the best schedule is a min of 2 max quantities:
Max (T_B, λ_A) & Max (T_A, λ_B).

Pop [69] uses the heuristic obtained from PCP scheduling to bound the schedules in a typical branch and bound algorithm to get to the optimal schedule. But branch and bound algorithm is an exponentially complex algorithm in the worst-case. So there is a need for a

lesser complex algorithm that can produce near-optimal schedules. From a higher view point of scheduling we need to limit the need for BB scheduling as much as possible.
Initially, the control variables in the CDFG are extracted. Let c1, c2,.....,cn be the control variables. Then there will be at most 2^n possible data-flow paths of execution for each combination of these control variables from the given CDFG. An ideal aim is to get the optimal schedule at compile time for each of these paths. Since the control information is not available at compile time, we need to arrive at an optimal solution for each path with every other path in mind. This optimal schedule is arrived at in two stages. First the optimal individual schedule for each path is determined. Then each of these optimal schedules is modified with the help of other schedules.

Stage 1: There are $m=2^n$ possible Data Flow Graphs (DFG's). For each DFG, the PCP scheduling is done. Then, the DFG's are ordered in the decreasing order of their total delays. An optimal solution can be obtained by doing branch and bound scheduling for each of these PCP scheduled DFG's. But branch and bound is a highly complex algorithm with exponential complexity. In this case, this complex operation needs to be done 2^n times, where n is the number of control variables, which increases the complexity way beyond control. Hence Branch and bound is done only when it is essential to do so. Then BB scheduling is done for DFG1, which has the largest delay. For DFG2, the PCP delay is compared with the BB delay of DFG1. If the PCP delay is smaller, then the PCP scheduling is taken as the optimal schedule for that path. If not, then the BB scheduling is done to get the optimal schedule. It makes sense to do this, as the final delay of each DFG after modification is going to be close to the delay of the worst delay path. In the same way, the optimal schedule is arrived at for each of the DFG.

Stage 2: Once the optimal schedule is arrived at, a schedule table is initialized with the processes on the rows and the various combinations of control variables on the column. A branching tree is also generated, which shows the various control paths. This contains only the control information of the CDFG. There exists a column in the schedule table corresponding to each path in this branching tree. The branching tree is shown in Figure 16. The path corresponding to the maximum delay is taken and the schedule for that corresponding path is taken as the template (DCK'). Now the DCK path is taken and the schedule is modified according to that of DCK'. This is done for all the paths. The final schedule table obtained will be the table that resides on the processor.

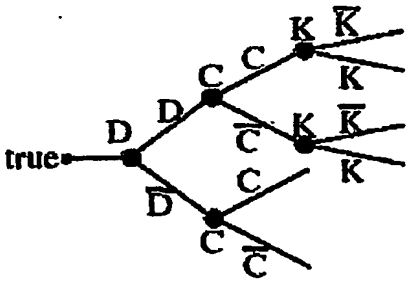


Figure 16: Branching Tree

The pseudo code of this process is summarized in Appendix G.

We also observe that processes with large execution times have a greater impact on the schedule than the shorter processes. Hence, we decided to schedule large processes in a special way. The shorter processes can be scheduled using the PCP scheduling algorithm. Since PCP scheduling is done for most of the processes, the complexity stays closer to O(N), where N is the number of processes to be scheduled.

a) Identify the first set of processes that need to be scheduled onto the same processor which are computationally complex. Let's call them MP1, MP2....(Macro process 1 etc.)
b) Schedule all the processes till these macro processes in the data flow graph using PCP scheduling.
c) Calculate the estimated execution time of the smaller processes to find the start time of each of the macro process.
d) Determine the next set of such macro processes in the DFG. Let's call them MP_sub1, MP_sub2...
e) For processes amidst these two sets of macro processes, PCP scheduling is used.
f) For processes occurring after the second set of macro processes, the execution times are added up to get the total execution time.
g) Now, determine the order of execution of these processes by estimating the worst-case execution time in each case and selecting the best amongst them.
h) After this scheduling, the block after the second set of macro processes is taken as the current DFG and steps a-g re implemented.
i) Step h is repeated till the end of DFG is reached.

Schedule merging:
In the schedule table there are some columns representing paths that are complete and some that are not. The incomplete paths can be now referred to as parent paths of possible complete paths.
In the example shown in Figure 13, we see that for earliest evaluation of all conditional variables (viz. D, C, K) it is necessary to evaluate D first, then C and then K. Therefore the tree of possible paths is as shown in Figure 16. Now, while creating the schedule table, initially only consider the full possible paths i.e. , the 6 paths listed in Figure 16. Perform scheduling by the suggested algorithm. This will fill these columns. Then create the remaining column of partial paths (i.e., D, D^C, ...etc). These are now just empty columns. Now if a process has the same start times in multiple columns, then push it into the parent empty column. This approach tries to obtain the worst case delay and merge all paths to that timeline. Since the D^C^K(bar) path had the worst case optimal delay, all other full paths were adjusted to match this path. But it is also necessary to consider the probability of the occurrence of all the full paths (6 of them). Then prune out the bottom 10% of the paths, that is, disregard those full paths whose probability of occurrence is less than a threshold value when compared to the path with most probable occurrence.
Then a path is selected from the remaining ones, whose probability of occurrence is the highest. This will be the new reference to which all the remaining paths will adjust to. Now it is likely that these chosen full paths and the disregarded full paths, share certain partial

paths (parent paths). Therefore, while allocating the start times for the processes that fall under these shared partial paths, we must allocate them based on the worst (most delay consuming) disregarded path which needs (shares) these processes. While performing schedule merging, all data dependencies must be respected.

4.3.2.3 Reconfiguration

In the discussion so far, we had not emphasized the need to involve reconfiguration times in the CDFGs. With an example we will show how this time can influence the tightness of a schedule. Consider the following task graph (Figure 17).

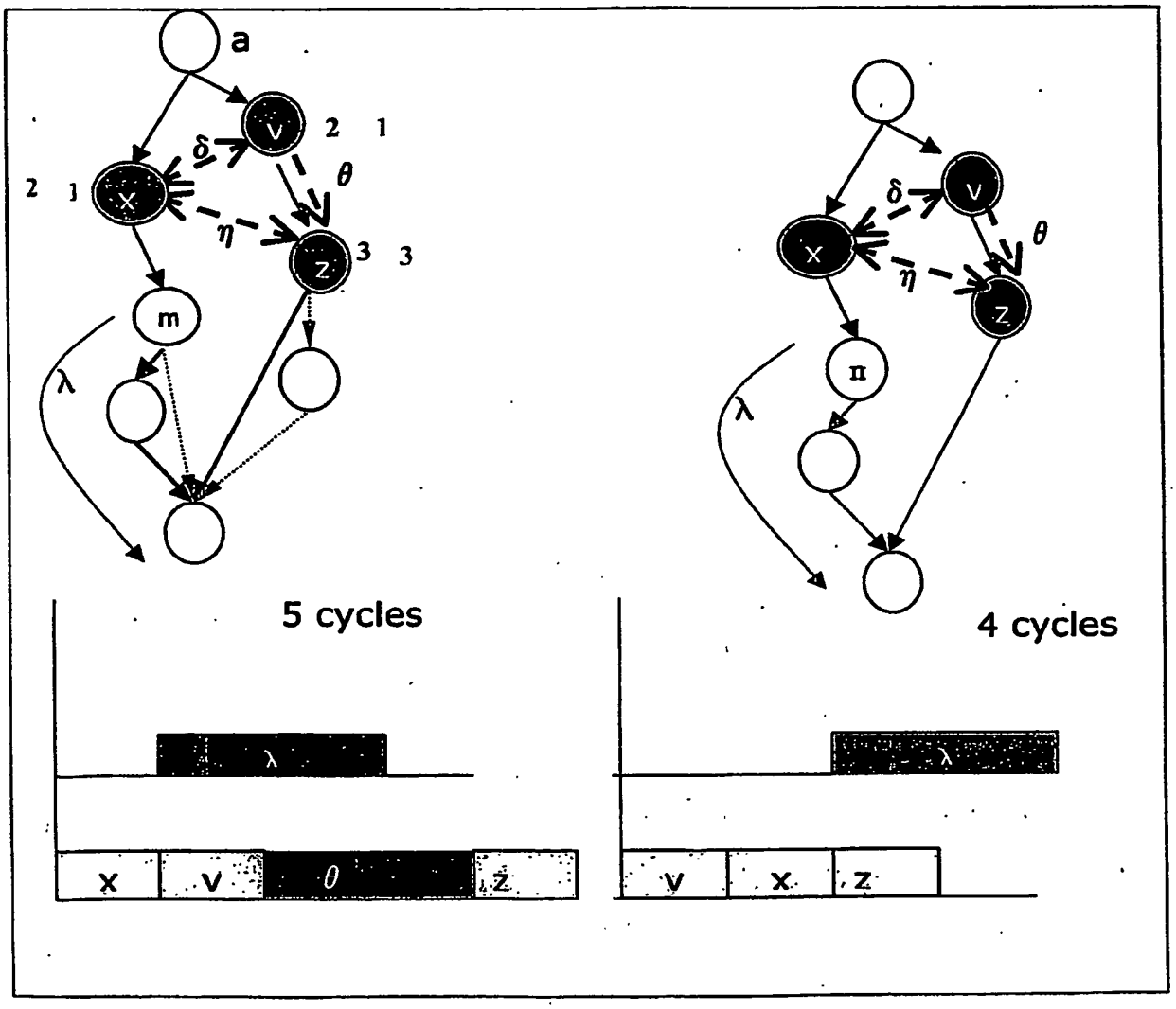


Figure 17: Influence of Reconfiguration time on Scheduling

In the task graph, say 'a' is a variable that influences the decision on which of the two mutually exclusive paths (red or green) will be taken, and a is known during run time but much earlier than 'm' & 'z' have started. Let x, v, z and λ be the times taken by processes in the event that 'a' happens to force the red path to be taken. Let processes x, v and z be mapped onto the same processing unit. Let θ, δ, η be the reconfiguration times for swapping between the processes on the unit. Given these circumstances, if run time scheduling according to [68] is applied, it neglects the reconfiguration times and provides a schedule of 5 cycles as shown on the left hand side. But if reconfiguration time were to have been considered, a schedule more like the one on the right hand side is tighter with 4 clock cycles. This simple example shows the importance of considering reconfiguration time in a reconfigurable processor, if fast swaps of tasks on the processing units need to be performed.

Therefore incorporating Reconfiguration time into Control flow graphs involves the following steps:

i. Special edges are added onto the control flow graphs. These graphs exist between a similar set of processes, which will be executed on the same processor with or without reconfiguration.
ii. Reconfiguration times affect the worst-case execution time of loopy codes. So this has to be taken care of, when loopy codes are being scheduled.
iii. Care needs to be taken to schedule the transfer of reconfiguration bit-stream from the main memory to the processor memory.

4.3.2.4 Loop-based scheduling

In static scheduling, loops whose iteration counts are not known at compile time impose scheduling problems on tasks which are data dependent on them, and those tasks that have resource dependency on their processing unit. Therefore, we have considered cases which are likely to impact the scheduling to the largest extent and provided solutions.

Case 1: Solitary loops with unknown execution time. Here, the problem is the execution time of the process is known only after it has finished executing in the processor. So static scheduling is not possible.

Solution:

(Assumption) Once a unit generates an output, this data is stored at the consuming / target unit's input buffer. Consider the following scheduled chart (Figure 18). Each row represents processes scheduled on a unique type of unit (Processor). Let P1 be the loopy process.

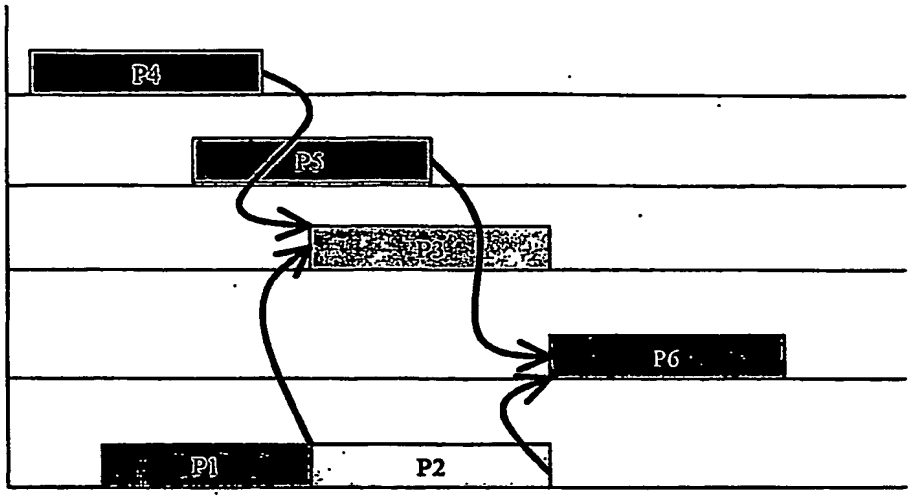


Figure 18: Scheduled Process Charts with Resource and Data Dependency

From the above figure we see that
P3 depends on P1 and P4,
P2 depends on P1,
P6 depends on P2 and P5.
If P1's lifetime exceeds the assumed lifetime (most probable lifetime or a unit iteration), then all dependents of P1 and their dependents (both resource and data) should be notified and the respective Network Schedule Manager (NSM) and Logic Schedule Manager (LSM) entries delayed. Of course, this implies that while preparing the schedule tables, 2 assumptions are made.

1) The lifetimes of solitary loops with unknown execution times are taken as per the most probable case obtained from prior trace file statistics (if available and applicable). Else unitary iteration is considered.
2) All processes that are dependent on such solitary loop processes are scheduled with a small buffer at their start times. This is to provide time for notification through communication channels about any deviation from assumption 1 @ run time.

If assumption 1 goes wrong, the penalty paid is:
Consider the example in Figure 15 where 2 processes in the ready list are being scheduled based on PCP. Now by PCP method if $\lambda_A > \lambda_B$ and P1 & P2 do not share the same resource, then PA is scheduled earlier than PB. WE have assumed that λ_A is due to most probable execution time of Loop P1. But at runtime if Loop P1 executes lesser # of times than predicted and therefore resulting in λ_A being $< \lambda_B$, then the schedule of PA earlier than PB results in being a mistake.
We calculate the time difference between both possible schedules. We do not at this point propose to repair the schedule because all processes before P1 have already been executed. And trying to fit another schedule at run time, requires intelligence on the communication network which is a burden. But on the brighter side, if @ run time Loop P1 executes more # of times than predicted, then λ_A will still be $> \lambda_B$. Therefore the assumed schedule holds true.

	Expression α	Expression β	Expression θ	Expression γ
Process A		0		
Process B		10		
.....				
.....				

	Expression α	Expression β	Expression θ	Expression γ
Process A		30		
Process B		40		
.....				
.....				

↓ .....

and so on.

Figure 19: Dynamic Entry Updates in the NSM and LSMs

Case 2: A combination of two loops with one loop feeding data to the other in an iterative manner.

Solution: Consider PA feeding data to PB in such a manner. For doing static scheduling, if we loop unroll them and treat it in a manner of smaller individual processes, then it is not

possible to assume unpredictable number of iterations. Therefore if unpredictable number of iterations is assumed in both loops, then memory foot-print could become a serious issue. But an exception can be made. If both loops at all times run for the same number of iterations, then the schedule table must initially assume either the most probable number of iterations or one iteration each and schedule PA,PB,PA,PB and so on in a particular column. In case the prediction is exceeded or fallen short off, then the NSM and LSMs must do 2 tasks:

1) If the iterations exceed expectations, then all further dependent processes (data and resource) must be notified for postponement and notified for scheduling upon the iterations completion with an appropriate difference in expected and obtained @ run time, schedule times. If the iterations fall short of expectations, then all further schedules must only be preponed.
2) Since the processes PA and PB should denote single iteration in the table, their entries should be continuously incremented @ run time by the NSM and the LSMs. The increment for one process of course happens for a predetermined number of times, triggered off by the schedule or execution of the other process. For example in Figure 19, we see that PA = 10 cycles, PB = 20 cycles and hence if both loops run for 5 times, then the entry in the column increments as shown.

Only in such a situation can there be preparedness for unpredictable loop iteration counts.

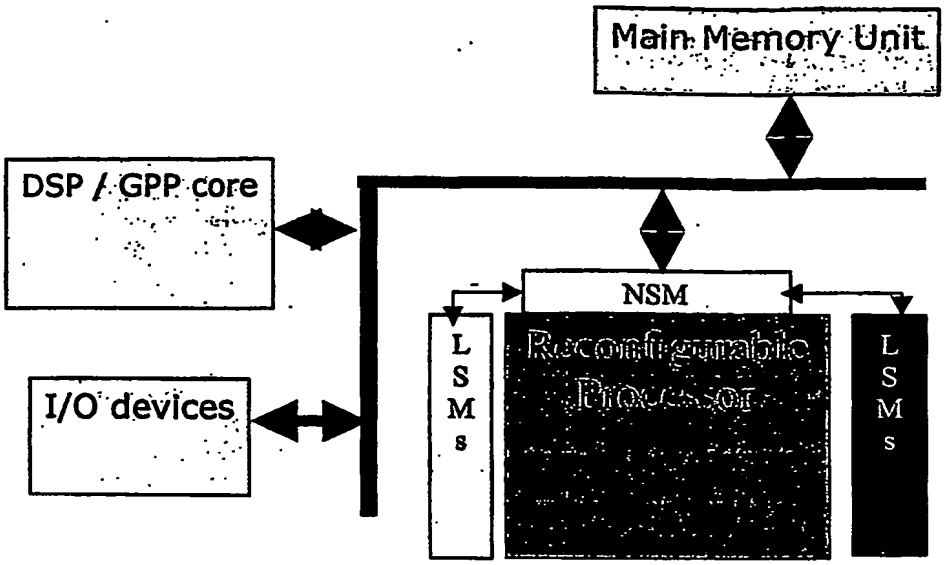
Case 3: A loop in the macro level i.e. containing more than a single process.

Solution: In this case, there are some control nodes inside a loop. Hence the execution time of the loop changes with each iteration. This is a much more complicated case than the previous options. Here lets consider a situation where there is a loop covering 2 mutually exclusive paths, each path consisting of 2 processes (A,B & C,D) with (3,7 & 15,5) cycle times. In the schedule table there will be a column to indicate an entry into the loop and 2 columns to indicate the paths inside the loop. Optimality in scheduling inside the loop can be achieved, but in the global scheme of scheduling, the solution is non-optimal. But this cannot be helped because to obtain a globally optimal solution, all possible paths have to be unrolled and statically scheduled. This results in a table explosion and is not feasible in situations where infinite number of entries in table are not possible. Hence, from a global viewpoint the loop and all its entries are considered as one entity with the most probable number of iterations considered and the most expensive path in each iteration is assumed to be taken. For example in the above case, path C,D is assumed to be taken all the time.

Now, a schedule is prepared for each path and hence entered into the table under 2 columns. When one schedule is being implemented, the entries for both columns in the next loop iteration is predicted by adding the completion time of the current path to both column entries (of course while doing this care should be taken not to overwrite the entries of the current path while they are still being used). Then when the current iteration is completed and a fresh one is started, the path is realized and the appropriate (updated / predicted) table column is chosen to be loaded form the NSM to the LSMs.

4.4 Network architecture

In order to coordinate the mapping of portions of the schedule table onto corresponding CLUs, we propose the following architecture. In Figure 20, the interfacing of the Reconfigurable unit with the host processor and other I/O and memory modules is shown.



LSM = Logic Schedule Manager; NSM = Network Schedule

Figure 20: Overview of the System Architecture

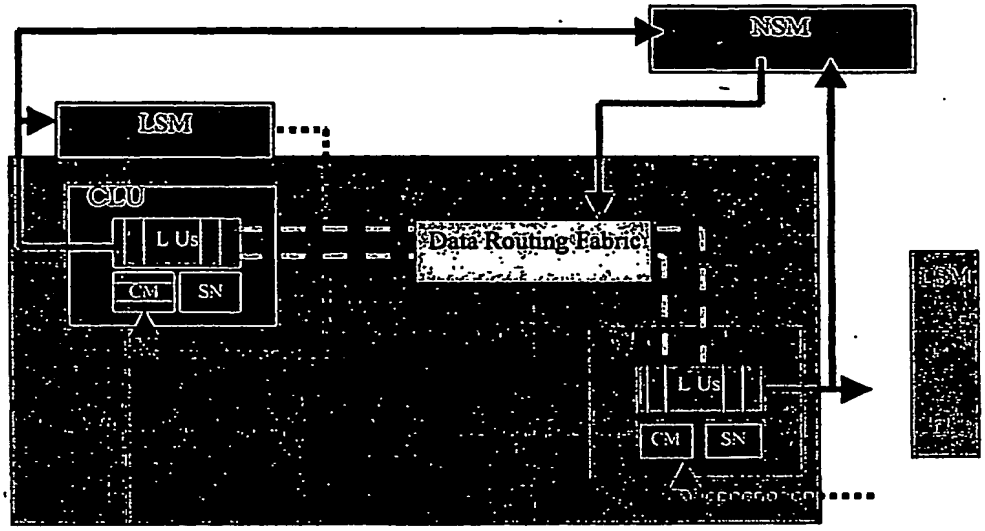
The Network Schedule Manager (Figure 21) has access to a set of tables, one for each processor. A table consists of possible tentative schedules for processes or tasks that must be mapped onto the corresponding processor subject to evaluation of certain conditional control variables. The Logic Schedule manager schedules and loads the configurations for the processes that need to be scheduled on the corresponding Processor ie. all processes that come in the same column (a particular condition) in the schedule table. In PCP scheduling, since the scheduling of the processes in the ready list depends only on the part of the paths following those processes, the execution time of the processes shall initially conveniently include the configuration time.

Once a particular process is scheduled and hence removed from the ready list, another process is chosen to be scheduled based on the pcp criteria again. But this time the execution time of that process is changed or rather reduced by using the reconfiguration time, instead of the configuration time. Essentially, for the first process that is scheduled in a column,

the *completion time = execution time + configuration time*

For the next or successive processes,

completion time = predecessor's completion time + execution time + reconfiguration time



CLU = Configurable Logic Unit; LU = Logic Units; SN = Switching Network
CM = Configuration Memory; LSM = Logic Schedule Manager

Figure 21: The Internals of the Reconfigurable Unit

Assuming that once a configuration has been loaded into the CM, the process of putting in place the configuration is instantaneous, it is always advantageous to load successive configurations into the CM ahead of time. This will mean a useful latency hiding for loading a successive configuration.

The reconfiguration time is dependent on two factors:

1) How much configuration data needs to be loaded into the CM (Application dependent)
2) How many wires are there to carry this info from the LSM to the CM (Architecture dependent)

The Network Schedule Manager should accept control parameters from all LSMs. It should have a set of address decoders because to send the configuration bits to the Network fabric consisting of a variety of switch boxes, it needs to identify their location. Therefore for every column in the table, the NSM needs to know the route apriori. We must NOT try to find a shortest path at run time. For a given set of processors communicating, there should be a fixed route. If this is not done then, the communication time of the edges n the CDFG cannot be used as constants while scheduling the graph.

For any edge the,

communication time = a constant and uniform configuration time

+

data transaction time.

The Network architecture consists of switch boxes and interconnection wires. The architecture will be based on the architecture described in [1]. This will be modeled as a combination of "Behavioral" & "Structural" style VHDL. Modifications that will be made are:

a. The Processing Elements derived in section 3 will be used instead of the 4 input LUTs that were used in Andre's model.
b. RAM style address access will be used to select a module or a switch box on the circuit.

c. Switch connections that are determined to be fixed for an application will be configured only once (at the start of that application).
d. Switch connections that are determined to be fixed for all applications will be shorted and the RC model for power consumption for that particular connection will be ignored for power consumption calculations.
e. The # of hierarchy levels will be determined by the application that has the maximum # of modules, because there is a fixed number of modules that can be connected

There will be 1 Network Schedule Manager (NSM) modeled in "Behavioral" & "Structural" style VHDL will store the static schedule table for the currently running application. The NSM collects the evaluated Boolean values of all conditional variables from every module.
For placing modules on the network 2 simple criteria are used. These are based on the assumption that the network consists of Groups of 4 Processing Unit Slots (G4PUS) connected in a hierarchical manner.
Note: A loop could include 0 or more number of CGPEs.
Therefore the following priority will be used for mapping modules onto the G4Pus:

a. A collection of 1 to 4 modules which are encompassed inside a loop shall be mapped to a G4PUS.
   i. If there are more than 4 modules inside a loop, then the next batch of 4 modules are mapped to the next (neighboring) G4PUS.
   ii. If # of CGPEs in a loop ≥ 2, then they will have greater priority over any FGPEs in that loop for a slot in the G4PUS.
b. For all other modules:
   iii. CGPE Modules with more than 1 Fan-in from other CGPEs will be mapped into a G4PUS.
   iv. CGPE Modules with more than 1 Fan-in from other FGPEs will be mapped into a G4PUS.

Note: The priorities are based on the importance for amount of communication between modules. Both Fan-ins and Fan-outs can be considered, for simplicity, we choose Fan-ins to CGPEs only.

5 Testing Methodology

In this research effort, we will focus mainly on reducing the number of reconfigurations that need to be made for running an application and then running other applications on the same processor. We also aim to reduce the time required to load these configurations from memory in terms of the number of configuration bits corresponding to the number of switches.
Time to execute an application for a given area (area estimate models of XILINX FPGAs and Hierarchical architectures can be used for only the routing portion of the circuit.) and a given clock frequency can be measured by simulation in VHDL.
The Time taken to swap clusters within an application and swap applications (reconfigure the circuit from implementing one application to another) is dependent on the similarity between the successor and predecessor circuits. We will measure the time to make a swap, in terms of # of bits required for loading a new configuration. Since a RAM style loading of configuration bits will be used, it is proven [2] to be faster than serial loading (used in Xilinx FPGAs). We expect speed up over the RAM style due to 2 reasons:

c) The address decoder can only access one switch box at a time. So the greater the granularity of the modules, the fewer the number of switches used and hence configured.

d) Compared to peer architectures which have only LUTs or a mixture of LUTs and CPGEs with low granularity (MAC units), we expect to have CGPEs of moderate granularity for abstract control-data flow structures in addition to FGPEs. Since these CPGEs are derived from the target applications, we expect their granularity to be the best possible choice for a reconfigurable purpose. They are modeled in "Behavioral" VHDL and are targeted to be implemented as ASICs. This inherently would lead to a reduced amount of configurations.

The Time taken to execute each application individually will be compared to available estimates obtained for matching area and clock specifications from work carried out by other researchers. This will be in terms of number of configurations per application, number of bits per configuration, number of configurations for a given set of applications and hence time in seconds for loading a set of configurations.

Note on power consumption: Sources of Power consumption for a given application can be classified into 4 parts:

b. Network power consumption due to configurations with an application. This is due to the Effective Load Capacitance on a wire for a given data transfer from one module to another for a particular configuration of switches.
   Note: The more the number of closed switches a signal has to pass through, the more the effective load capacitance and resistance. Shorted switches will not be considered to contribute to this power.
c. Data transfer into and out of the Processor
   Note: This can have a significant impact on the total power in media rich or communication dominated applications ported onto any processing platform.
d. Processing of data inside a module.
   Note: This will require synthesizable VHDL modules. But since our focus in this research work is on reducing power due to reconfiguration, we will leave this for future work.
e. The Clock distribution of the processor.
   Note: This can be measured if the all parts of the circuit are synthesizable. But we are focusing on a modeling aspect and do not consider this measurement.

At the level of modeling a circuit in VHDL, it is possible to only approximately determine the power consumptions. We will use the RC models of XILINX FPGAs and [1] architectures to get approximate power estimates. Power aware scheduling and routing architecture design are complex areas of research in themselves and is not the focus of this research effort. In this thesis we focus on reducing the amount of reconfigurations, which directly impacts the speed of the processor and indirectly impacts the power consumption to a certain extent.

We will compare the performance of our processor for each of the applications with available estimates (those published in literature) in terms of Time (execution time for an application at a given clock frequency).
We cannot compare the area parameter because the architecture being used is Rent's Hierarchical Model which was proposed in [1]. An optimized architecture for a given set of applications is beyond the scope of this thesis. Although the modules are customized based on the clustering approach, yet they are not in a synthesized form and hence their power consumption and area occupied are not determinable with the existing EDA tools.

6 Conclusions & Ongoing Research

After a detailed analysis of the current approaches towards designing a dynamic and fast reconfigurable processor, we have proposed a methodology consisting of an automated approach towards identifying reconfigurable regions in applications. We have also proposed a criterion for selection of the number of processing units of a given type to extract the maximum amount of resource utilization. Thereafter a scheduling strategy has been discussed that maps the tasks onto the computing resources on the processor. We have proposed to select a collection of algorithms from areas of media processing and computer graphics to test the methodology. Currently work is in progress in refining the individual algorithms for graph matching and scheduling. With the completion of the set of tools proposed, an automated approach towards developing dynamically reconfigurable processors will be available.

7 References

1. Andre Dehon. "Reconfigurable architectures for general purpose computing," Ph.D Thesis, MIT, 1996.
2. Varghese George and Jan M. Rabaey. "Low –Energy FPGAs – Architecture and Design," Kluwer Academic Publishers.
3. M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek. "Object Oriented Circuit-Generators in Java," IEEE Symposium on FPGAs for Custom Computing Machines, April 1998.
4. Ryan Kastner, Seda Ogrenci Memik, Elaheh Bozorgzadeh and Majid Sarrafzadeh. "Instruction Generation for Hybrid Reconfigurable Systems," International Conference on Computer-Aided Design (ICCAD), November, 2001
5. Philip Brisk, Adam Kaplan, Ryan Kastner and Majid Sarrafzadeh. "Instruction Generation and Regularity Extraction for Reconfigurable Processors," International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), October 2002
6. W. Lee, R. Barua, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. "Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine," Proc of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS), San Jose, CA, October, 1998.
7. Anant Agarwal, Saman Amarasinghe, Rajeev Barua, Matthew Frank, Walter Lee, Vivek Sarkar, Devabhaktuni Srikrishna and Michael Taylor. "The Raw Compiler Project," Proc of the Second SUIF compiler workshop, Stanford, CA, August 21-23, 1997.
8. A. DeHon. "The Density Advantage of Configurable Computing," Computer, vol. 33, no. 4, April 2000, pp.41-49
9. R. Reed Taylor and Seth Copen Goldstein. "A High-Performance Flexible Architecture for Cryptography," Proc of the Workshop on Cryptographic Hardware and Embedded Systems, 1999.
10. Moreno, J.M, Cabestany, J. et al. "Approaching evolvable hardware to reality: The role of dynamic reconfiguration and virtual meso-structures," Microelectronics for Neural, Fuzzy and Bio-Inspired Systems, 1999.
11. Kiran Kumar Bondalapati. "Modeling and mapping for dynamically reconfigurable hybrid architectures," Ph.D Thesis, USC, 2001

12. Mirsky, E. DeHon, A. "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," IEEE Symposium on FPGAs for Custom Computing Machines, 1996.
13. Vorbach, M. Becker, J. "Reconfigurable Processor Architectures for Mobile Phones," Proc of International on Parallel and Distributed Processing Symposium, 2003.
14. Ebeling, C.Cronquist et al. "Mapping applications to the RaPiD configurable architecture," The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines, 1997.
15. Callahan, T.J. Hauser, J.R. Wawrzynek, J. "The Garp architecture and C compiler," IEEE Transactions on computers, 2000.
16. Singh, H. Ming-Hau Lee Guangming Lu Kurdahi, F.J. Bagherzadeh, N. Chaves Filho, E.M. "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," IEEE Transactions on computers, 2000.
17. Tsukasa Yamauchi et al. "SOP: A reconfigurable massively parallel system and its control-data-flow based compiling method," IEEE Symposium on FPGAs for Custom Computing Machines, 1996.
18. Scott Hauck et al. "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," International Conference on Computer Architecture, 2000.
19. P.M. Athanas and H.F. Silverman. "An Adaptive Hardware Machine Architecture for Dynamic Processor Reconfiguration," International Conference on Computer Design, 1991.
20. Peter M. Athanas. "A functional reconfigurable architecture and compiler," Technical Report LEMS-100, Brown University, Division of Engineering, 1992.
21. S. Sawitzki and A. Gratz and R. Spallek. "CoMPARE: A Simple Reconfigurable Processor Architecture Exploiting Instruction Level Parallelism," Proc. of PART, pp. 213--224, Springer-Verlag, 1998.
22. Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin and Brad Hutchings. "A Reconfigurable Arithmetic Array for Multimedia Applications," Proc of the ACM/SIGDA seventh international symposium on Field programmable gate arrays, 1999.
23. E.Sanchez, C.Iseli. "A C++ compiler for FPGA custom execution units synthesis," IEEE Symposium on FPGAs for Custom Computing Machines, 1995.
24. Bernardo Kastrup, Arjan Bink, Jan Hoogerbrugge. "ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator," IEEE Symposium on Field programmable Custom Computing Machines, 1999.
25. Michael Bedford Taylor; Anant Agarwal. "Design Decisions in the Implementation of a Raw Architecture Workstation," MS Thesis, MIT, 1996.
26. Hartenstein, R. Herz, M. Hoffmann, T. Nageldinger, U. "KressArray Xplorer: a new CAD environment to optimize reconfigurable datapath array architectures," Proc of the ASP-DAC Asia and South Pacific Design Automation Conference, 2000.
27. Bittner, R.A., Jr. Athanas, P.M. "Computing kernels implemented with a wormhole RTR CCM," The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines, 1997.
28. Miyamori, T. Olukotun, U. "A quantitative analysis of reconfigurable coprocessors for multimedia applications," IEEE Symposium on FPGAs for Custom Computing Machines, 1998.
29. Becker, J. Pionteck, T. Habermann, C. Glesner, M. "Design and implementation of a coarse-grained dynamically reconfigurable hardware architecture," IEEE Computer Society Workshop on VLSI, 2001.
30. www.broadcom.com

31. George, V. Hui Zhang Rabaey, J. "The design of a low energy FPGA," International Symposium on Low Power Electronics and Design, 1999.
32. Chen, D.C. Rabaey, J.M. "A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths," IEEE Journal of Solid-State Circuits, 1992.
33. Marlene Wan; Jan Rabaey et al. "Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System," Journal of VLSI Signal Processing, 2000.
34. Campi, F.Cappelli, A. et al. "A reconfigurable processor architecture and software development environment for embedded systems," International Parallel and Distributed Processing Symposium, 2003.
35. Jack Liu, Fred Chow, Timothy Kong, and Rupan Roy. "Variable Instruction Set Architecture and Its Compiler Support," IEEE Transactions on computers, 2003.
36. Marco Jacobs, Ivan Greenberg and Mike Strauss. "BOPS: Conquering the Geometry Pipeline," Game Developers Conference. March 22-26, 2004, San Jose. California
37. Brian Schoner, Chris Jones and John Villasenor. "Issues in Wireless Video Coding usign Run-time-reconfigurable FPGAs," Proc of the IEEE Symposium on FPGAs for Custom Computing Machines, Napa CA, April 19-21 1995.
38. Abbas Ali Mohamed, Szirmay-Kalos László, Horváth Tamás. "Hardware Implementation of Phong Shading using Spherical Interpolation," Periodica Polytechnica, Vol. 44, Nos 3-4, 2000.
39. D. A. Basin. "A term equality problem equivalent to graph isomorphism. Information Processing Letters," 54:61–66, 1994.
40. M. R. Garey and D. S. Johnson. "Computers and Intractability: A Guide to the Theory of NP-Completeness," W. H. Freeman, New-York, 1979.
41. J. E. Hopcroft and J. K. Wong. "Linear time algorithm for isomorphism of planar graphs," Sixth ACM Symposium on Theory of Computing, 1974.
42. S. W. Reyner. "An analysis of a good algorithm for the subtree problem," SIAM Journal of Computing, 6(4):730–732, 1977.
43. A. M. Abdulkader. "Parallel Algorithms for Labelled Graph Matching," PhD thesis, Colorado School of Mines, 1998.
44. B. T. Messmer and H. Bunke. "A decision tree approach to graph and subgraph isomorphism detection," Pattern Recognition, 32:1979–1998, 1999.
45. Michihiro Kuramochi and George Karypis, "An Efficient Algorithm for Discovering Frequent Subgraphs," Technical Report 02-026. University of Minnesota
46. K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching," Proc. of Design Automation Conference, 1987.
47. A. Chowdhary, S. Kale, P. Saripella, N. Sehgal and R. Gupta. "A General Approach for Regularity Extraction in Datapath Circuits," Proc. of International Conference on Computer-Aided Design, 1998.
48. D. S. Rao and F. J. Kurdahi. "On Clustering for Maximal Regularity Extraction," IEEE Trans. on Computer-Aided Design, Vol. 12, No. 8, August, 1993.
49. S. Cadambi and S. C. Goldstein. "CPR: A Configuration Profiling Tool," Proc. of the Symposium on Field-Programmable Custom Computing Machines, 1999.
50. S. Gold and A. Rangarajan. "A graduated assignment algorithm for graph matching," IEEE Transactions on Pattern Analysis and Machine Intelligence, 18(4):377–88, 1996.
51. S.-J. Farmer. "Probabilistic graph matching," University of York, 1999.
52. A. Perchant and I. Bloch. "A new definition for fuzzy attributed graph homomorphism with application to structural shape recognition in brain imaging," In IMTC'99, 16th IEEE

Instrumentation and Measurement Technology Conference, pages 1801–1806, Venice, Italy, May 1999.
53. J. Sung Hwan. "Content-based image retrieval using fuzzy multiple attribute relational graph," IEEE International Symposium on Industrial Electronics Proceedings (ISIE 2001), 3:1508–1513, 2001.
54. C.-W. K. Chen and D. Y. Y. Yun. "Unifying graph-matching problem with a practical solution," In Proceedings of International Conference on Systems, Signals, Control, Computers, September 1998
55. Anand Rangarajan and Eric Mjolsness, A Lagrangian. "Relaxation Network for Graph Matching," IEEE Transactions on Neural Networks, 7(6):1365-1381, 1996
56. Kimmo Fredriksson. "Faster string matching with super—alphabets," Proc of SPIRE'2002, Lecture Notes in Computer Science 2476, pages 44-57, Springer Verlag, Berlin 2002.
57. Ganesh Lakshminarayana, Kamal S. Khouri, Niraj K. Jha, Wavesched. "A Novel Scheduling Technique for Control-Flow Intensive Designs," IEEE Transactions On Computer-Aided Design Of Integrated Circuits And Systems, Vol. 18, No. 5, May 1999
58. D. D. Gajski, N. Dutt, A. Wu, and S. Lin, High-Level Synthesis. "Introduction to Chip and System Design," Boston, MA: Kluwer Academic, 1992.
59. W. Wolf, A. Takach, C. Huang, and R. Mano. "The Princeton university behavioral synthesis system," Proc. Design Automation Conf., June 1992, pp. 182–187.
60. D. Ku and G. De Micheli. "Relative scheduling under timing constraints,". IEEE Trans. Computer-Aided Design, vol. 11, pp. 696–718, June 1992.
61. C. Chekuri, Richard Johnson, Rajeev Motwani, Balas Natarajan, Bob Rau, and Michael Schlansker. "An Analysis of Profile-Driven Instruction Level Parallel Scheduling with Application to Super Blocks," Proc of the 29th Annual International Symposium on Microarchitecture (MICRO-29), December 1996.
62. J. A. Fisher. "Global code generation for instruction level parallelism," Tech. Rep. HPL-93-43, Hewlett Packard Labs, June 1993.
63. W.W. Hwu et al. "The super block: An effective technique for VLIW and superscalar compilation," Journal. of Supercomputing, 7:229–248 (1993).
64. J.C. Dehnert and R.A. Towle. "Compiling for the Cydra-5," Journal of Supercomputing, 7:181-228, (1993).
65. Hesham L. Rewini and Hesham H. Ali. "Static scheduling of conditional branches in parallel programs," Journal of Parallel and Distributed Computing, 24(1): 41-54, January 1994.
66. Lin Huang and Michael J. Oudshroon. "An approach to distribution of parallel programs with conditional task attributes," Technical Report TR97-06, Department of Computer Science, University of Adelaide, August 1997.
67. Ling Huang, Michael J. Oudshroon and Jiannong Cao. "Design and implementation of an adaptive task mapping environment for parallel programming," Australian Computer Science Communications, 19(1):326 – 335, February 1997.
68. V. Mooney. "Path-Based Edge Activation for Dynamic Run-Time Scheduling," International Symposium on System Synthesis (ISSS'99), pp. 30-36, November 1999.
69. Petru Eles, Alex Doboli, Paul Pop, Zebo Peng. "Scheduling with Bus Access Optimization for Distributed Embedded Systems," IEEE Trans on VLSI Systems, vol. 8, No 5, 472-491, October 2000.
70. E.G. Coffman Jr., R.L. Graham. "Optimal Scheduling for two Processor Systems," Acta Informatica, 1, 1972, 200-213.

71. H. Kasahara, S. Narita. "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," IEEE Trans. On Comp., V33, N11, 1984, 1023-1029.
72. Y.K. Kwok, I. Ahmad. "Dynamic Critical-Path Scheduling: an Effective Technique for Allocating TaskGraphs to Multiproces-sors," IEEE Trans. on Parallel and Distributed Systems, V7, N5, 1996, 506-521.
73. P. Chou, G. Boriello. "Interval Scheduling: Fine-Grained Code Scheduling for Embedded Systems," Proc. ACM/IEEE DAC, 1995, 462-467.
74. R. K. Gupta, G. De Micheli. "A Co-Synthesis Approach to Embedded System Design Automation," Design Automation for Embedded Systems, V1, N1/2, 1996, 69-120.
75. F. R. Brown III. "Real-Time Scheduling with Fuzzy Systems," PhD thesis, Utah State University, 1998.
76. Y. Jiajun, X. Guodong, C. Xibin, and M. Xingrui. "A fuzzy expert system architecture implementing onboard planning and scheduling for autonomous small satellite," 12th Annual AIAA/ Utah State University Conference on Small Satellites, Logan, Utah, Aug. 1998.
77. A. Dasu. "The need for reconfigurable multimedia processing," Ph.D. qualifying report. 2001.

8 Appendix A

A Control Data Flow Graph consists of both data flow and control flow portions. In compiler terminology, all regions in a code that lie in between branch points are referred to as Basic Blocks. Those basic blocks which have additional code due to code movement, shall be referred to these as zones because. Also under certain conditions, decision making control points can be integrated into the basic block regions. These blocks should be explored for any type of data level parallelism they have to offer. Therefore for simplicity in the following description, basic blocks are referred to as zones. The methodology remains the same when modified basic blocks and abstract structures such as nested loops and hammock structures etc are considered as zones.

High level ASNI C code of the target application is first converted to an assembly code (UltraSPARC). Since the programming style is user dependent, the assembly code needs to be expanded in terms of all functions calls. To handle the expanded code, a suitable data structure that has a low memory footprint is utilized. Assembly instructions that act as delimiters to zones must then be identified. The data structure is then modified to lend itself to a more convenient form for extracting zone level parallelism.

The following are the steps involved in extracting zone level parallelism.

Step-1: Parsing the assembly files

In this step for each assembly (.s) file a doubly linked list is created where each node stores one instruction with operands and each node has pointers to the previous and next instructions in the assembly code. Parser ignores all commented out lines, lines without instructions except the labels such as

Main:

.LL3:

Each label starting with .LL is replaced with a unique number (unique over all functions)

Step-2: Expansion

Each assembly file that has been parsed is stored in a separate linked list. In this step the expander moves through the nodes of linked list that stores main.s. If a function call is detected that function is searched through all linked lists. When it is found, that function from the beginning to the end, is copied and inserted into the place where it is called. Then the expander continues moving through the nodes from where it stopped. Expansion continues until the end of main.s is reached. Note that if an inserted function is also calling some other function expander also expands it until every called function is inserted to the right place.

In the sample code (Appendix B), main() function is calling the findsum() function twice and findsum() function is calling the findsub() function. The expanded code (after considering individual assembly codes (Appendix C) is shown in Appendix-D.

Step-3: Create Control Flow Linked List

Once the main.s function has been expanded and stored in a doubly linked list, the next step is to create another doubly linked list (control_flow_linked_list) that stores the control flow information. This will be used to analyze the control flow structure of the application code, to detect the starting and ending points of functions and control structures (loops, if..else statements, etc.).

As the expanded linked list is scanned, nodes are checked if they belong to a:

- Label or
  Function or
- Conditional or
- unconditional branch

In which case, a new node is created to be appended to the control flow linked list by setting the member pointers as defined below.

If the current node is a

- **function label**

A pointer to the expanded list pointing to the function label node
A pointer to the expanded list pointing to the beginning of the function (the next node of the function label node)
A pointer to the expanded list pointing to the end of the function
And node type is set to "function".

- **label**

A pointer to the expanded list pointing to the function label node
A pointer to the expanded list pointing to the beginning of the label (the next node of the label node)
And node type is set to "square".

- **unconditional branch** (b)

A pointer to the expanded list pointing to the branch node
A pointer to the control flow linked list pointing to the node that stores the matching target label of the branch instruction.
And node type is set to "dot"

- **conditional branch** (bne, ble, bge, ...etc)

A pointer to the expanded list pointing to the branch node
A pointer to the control flow linked list pointing to the node that stores the matching target label of the branch instruction.
And node type is set to "circle".
The control flow linked list output for the findsum.s function is shown in Appendix D.

Step 4: Modification of Control Structure
The control structure linked list (which essentially represents the control flow graph of the candidate algorithm) is then modified as follows.

- The pointers from unconditional branch nodes (also called "dot" nodes) to the next node in the list need to be disconnected and made NULL. Hence for the "dot" node:
  node→ next = NULL
  for the following node:
  node→ previous = NULL
  {Exception: if the next node of the "dot" node is itself the target node !}

- The target nodes of the unconditional branches need to be marked as "Possible Exit" nodes. These "Exit" classes of nodes are a subset of the regular "Target" or "Square" nodes.
- If unconditional branch node's rank is higher than target node's rank (indicating a feed back or loop), disconnect the link and mark as NULL.
  Hence for the "dot" node:
  node→ to_target = NULL
  But before disconnecting, mark target→ next (which should be a circle) as "loop node".
- In a special case, if an unconditional branch and a square share the same node, then the target of that unconditional branch is declared as an exit square with a loop type (because, instructions following this square, comprise the meat of the do-while loop). This exit square, will not have its next→ pointing to a circle. The circle is accessed through the dot node using the previous→ pointer. Then it is marked off as type loop.
- If a "Possible Exit" node has 2 valid input pointers, and rank of both source pointers is lesser than the node in consideration, then it is an "Exit" node and, disconnect the link to the corresponding "dot" node, and hence also mark that "dot" node's target pointer to NULL. In other words, if the node→ previous pointer of the "square/target" node of the "dot" node does not point to the "dot" node, then it has 2 valid pointers.
  Hence for the "dot" node:
  node→ to_target = NULL

For a sample high level code in the Figure 1 below, following which is the expanded assembly file. The control flow linked list is as shown in Figure 2. After modifications to this linked list a structure as indicated in figure 3 is obtained.

```
#include<stdio.h>
void main()
{
int
i=0,j=0,k=0,l=0,m=0,n=0,p=0,r=0;

 for(i=1;i<10;i++)
 {
  p = p - 8;
  p = p* 7;
 }
 i = i+ 1;
 if(i==j)
 {
  n = 9;
  if (k>0)
  {
        p = 19;
  }
  else
  {
   r = 23;
  }
  n= 17 + 8;
else
 {
  l = 10;
  m = n +r;
 }
 k = k -14;
 k = 7 - 8 * p;
 while(i<p)
 {
  p = p * 20;
  p = p - 7;
  while(k == 8)
  {
   p = p+ 17;
   i = i * p;
  }
  p= p - 23;
 }
 m = m +5;
 n = n +4;
}
}
```

Figure 1: An Example Program

The gcc (version 2.95.2) compiled code for the UltraSPARC architecture with node labeling is as follows:

```
        .file   "loop_pattern4.c"
gcc2_compiled.:
        .global .umul
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc   020
main:
        !#PROLOGUE# 0
        save    %sp, -144, %sp
        !#PROLOGUE# 1
        st      %g0, [%fp-20]                   ground
        st      %g0, [%fp-24]
        st      %g0, [%fp-28]
        st      %g0, [%fp-32]
        st      %g0, [%fp-36]
        st      %g0, [%fp-40]
        st      %g0, [%fp-44]
        st      %g0, [%fp-48]
        mov     1, %o0
        st      %o0, [%fp-20]
.LL3:
        ld      [%fp-20], %o0                   square 3
        cmp     %o0, 9
        ble     .LL6                    circle 6
        nop
        b       .LL4                    dot 4
         nop
.LL6:
        ld      [%fp-44], %o0                   square 6
        add     %o0, -8, %o1
        st      %o1, [%fp-44]
        ld      [%fp-44], %o0
        mov     %o0, %o1
        sll     %o1, 3, %o2
        sub     %o2, %o0, %o0
        st      %o0, [%fp-44]
.LL5:
        ld      [%fp-20], %o0                   square 5
        add     %o0, 1, %o1
        st      %o1, [%fp-20]
```

```
        b       .LL3                    dot 3
        nop
.LL4:
        ld      [%fp-20], %o0                   square 4
        add     %o0, 1, %o1
        st      %o1, [%fp-20]
        ld      [%fp-20], %o0
        ld      [%fp-24], %o1
        cmp     %o0, %o1
        bne     .LL7                    circle 7
        nop
        mov     9, %o0
        st      %o0, [%fp-40]
        ld      [%fp-28], %o0
        cmp     %o0, 0
        ble     .LL8                    circle 8
        nop
        mov     19, %o0
        st      %o0, [%fp-44]
        b       .LL9                    dot 9
        nop
.LL8:
        mov     23, %o0                 square 8
        st      %o0, [%fp-48]
.LL9:
        mov     25, %o0                 square 9
        st      %o0, [%fp-40]
        b       .LL10                   dot 10
        nop
.LL7:
        mov     10, %o0                 square 7
        st      %o0, [%fp-32]
        ld      [%fp-40], %o0
        ld      [%fp-48], %o1
        add     %o0, %o1, %o0
        st      %o0, [%fp-36]
.LL10:
        ld      [%fp-28], %o0                   square 10
        add     %o0, -14, %o1
        st      %o1, [%fp-28]
        ld      [%fp-44], %o0
        mov     %o0, %o1
        sll     %o1, 3, %o0
        mov     7, %o1
        sub     %o1, %o0, %o0
        st      %o0, [%fp-28]
.LL11:
```

```
        ld      [%fp-20], %o0                   square 11
        ld      [%fp-44], %o1
        cmp     %o0, %o1
        bl      .LL13                   circle 13
        nop
        b       .LL12                   dot 12
         nop
.LL13:
        ld      [%fp-44], %o0                   square 13
        mov     %o0, %o2
        sll     %o2, 2, %o1
        add     %o1, %o0, %o1
        sll     %o1, 2, %o0
        st      %o0, [%fp-44]
        ld      [%fp-44], %o0
        add     %o0, -7, %o1
        st      %o1, [%fp-44]
.LL14:
        ld      [%fp-28], %o0                   square 14
        cmp     %o0, 8
        be      .LL16                   circle 16
        nop
        b       .LL15                   dot 15
         nop
.LL16:
        ld      [%fp-44], %o0                   square 16
        add     %o0, 17, %o1
        st      %o1, [%fp-44]
        ld      [%fp-20], %o0
        ld      [%fp-44], %o1
        call    .umul, 0
         nop
        st      %o0, [%fp-20]
        b       .LL14                   dot 14
         nop
.LL15:
        ld      [%fp-44], %o0                   square 15
        add     %o0, -23, %o1
        st      %o1, [%fp-44]
        b       .LL11                   dot 11
         nop
.LL12:
        ld      [%fp-36], %o0                   square 12
        add     %o0, 5, %o1
        st      %o1, [%fp-36]
        ld      [%fp-40], %o0
        add     %o0, 4, %o1
```

```
        st      %o1, [%fp-40]
.LL2:
        ret                                     square 2
        restore
.LLfe1:
        .size   main,.LLfe1-main
        .ident  "GCC: (GNU) 2.95.2 19991024 (release)"
```

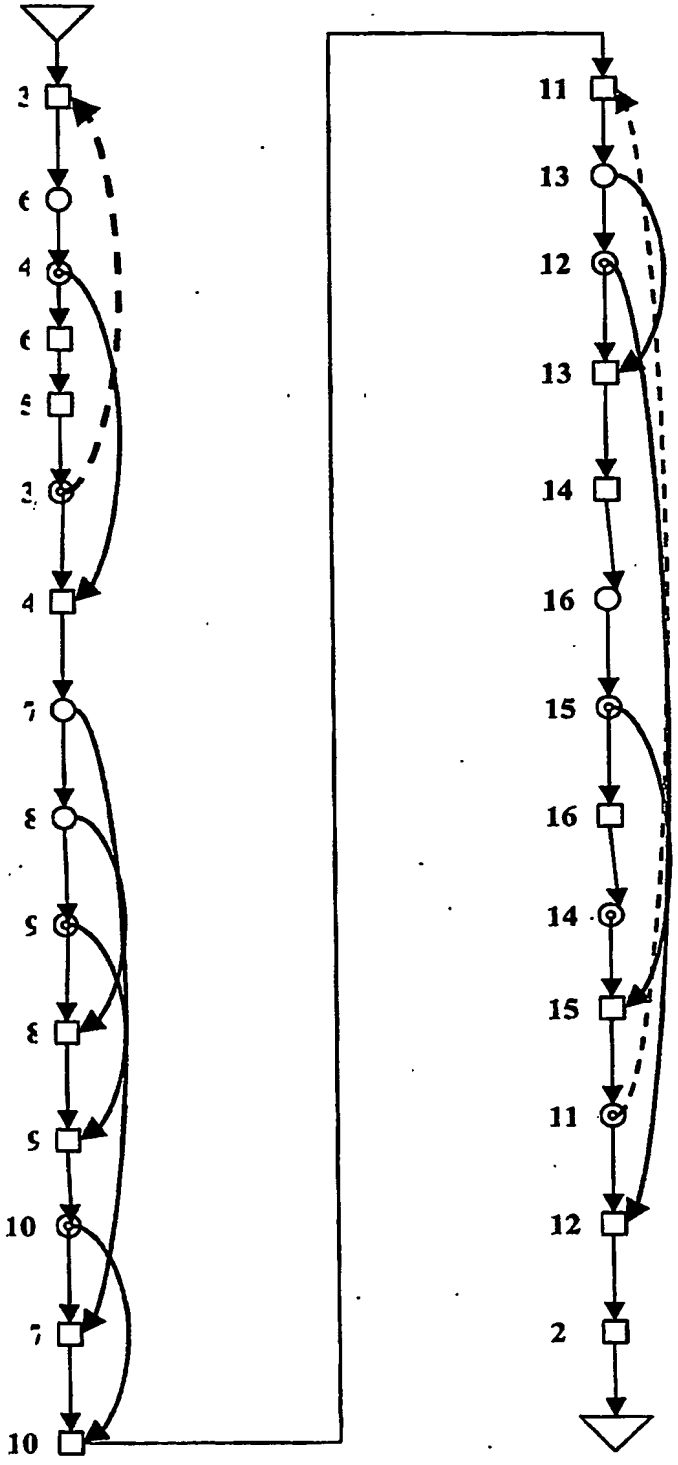


Figure 2: Control Flow Linked List

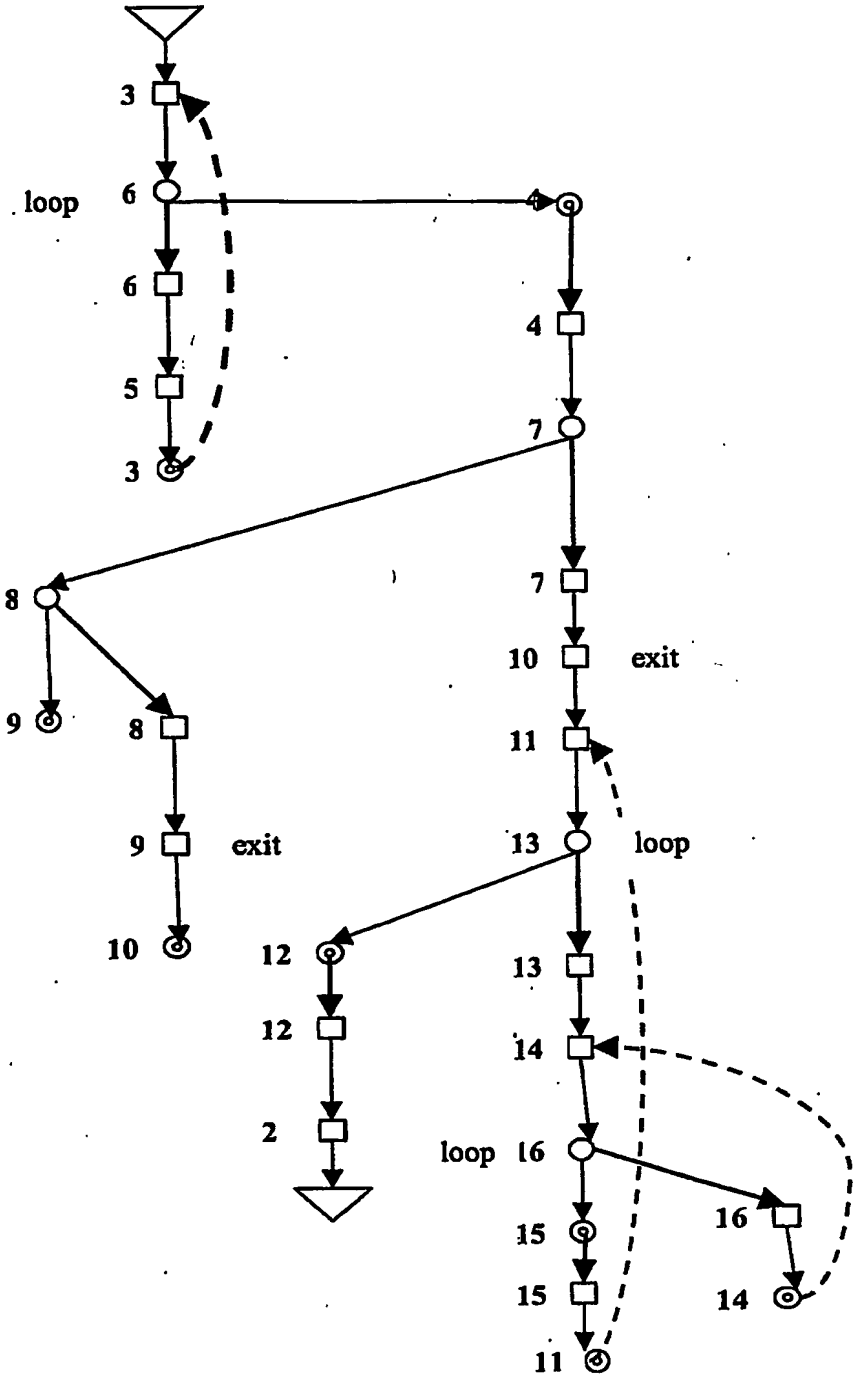


Figure 3: Modified Structure obtained from the Control Flow Linked List

Step 5: Creation of Zones

To extract all possibilities of parallelism and reconfiguration, zones are identified in the modified structure. But to identify such sections, delimiters are needed. A delimiter can be any of the following types of nodes:

(i) Circle
(ii) Dot
(iii) Exit square
(iv) Square
(v) Power
(vi) Ground.

A 'Circle' can indicate the start of a new zone or the end of a zone. A 'Dot' can only indicate the end of a zone or a break in a zone. An 'Exit square' can indicate the start of a new zone or the end of a zone. A 'Square' can only indicate the continuation of a break in the current zone. A 'Power' can only indicate the beginning of the first zone. A 'Ground' can only indicate the end of a zone.

Figure 4 shows example zones to illustrate the use of delimiters. Three zones, 1, 2, and 3 all share a common node, 'Circle 6'. This node is the end of Zone 1 and the start of zones 2 and 3. Zone 1 has the 'Power' node as its start, while Zone 6 has 'Ground' node as its end. The 'Dot 3' in Zone 3 indicates the end of that zone while 'Dot 4' indicates a break in Zone 2. This break is continued by 'Square 4'. In Zone 4, 'Square 9' indicates the end of the zone while it marks the start of Zone 5.

This function identifies zones in the structure, which is analogous to the numbering system in the chapter page of a book. Zones can have sibling zones (to identify if/else conditions, where in only one of the two possible paths can be taken {Zones 4 and 7 in Figure 1}) or child zones (to identify nested control structures {Zone 10 being child of zone 8 in Figure 1}). Zone types can be either simple or loopy in nature (to identify iterative loop structures). The tree is scanned node by node and decisions are taken to start a new zone or end an existing zone at key points such as circles, dots and exit squares. By default, when a circle is visited for the first time, the branch taken path is followed. But this node along with the newly started zone is stored in a queue for a later visit along the branch not taken path. When the structure has been traversed along the "branch taken" paths, the nodes with associated zones are popped out from the stack and traversed along their "branch not taken" paths. This is done till all nodes have been scanned and stack is empty.

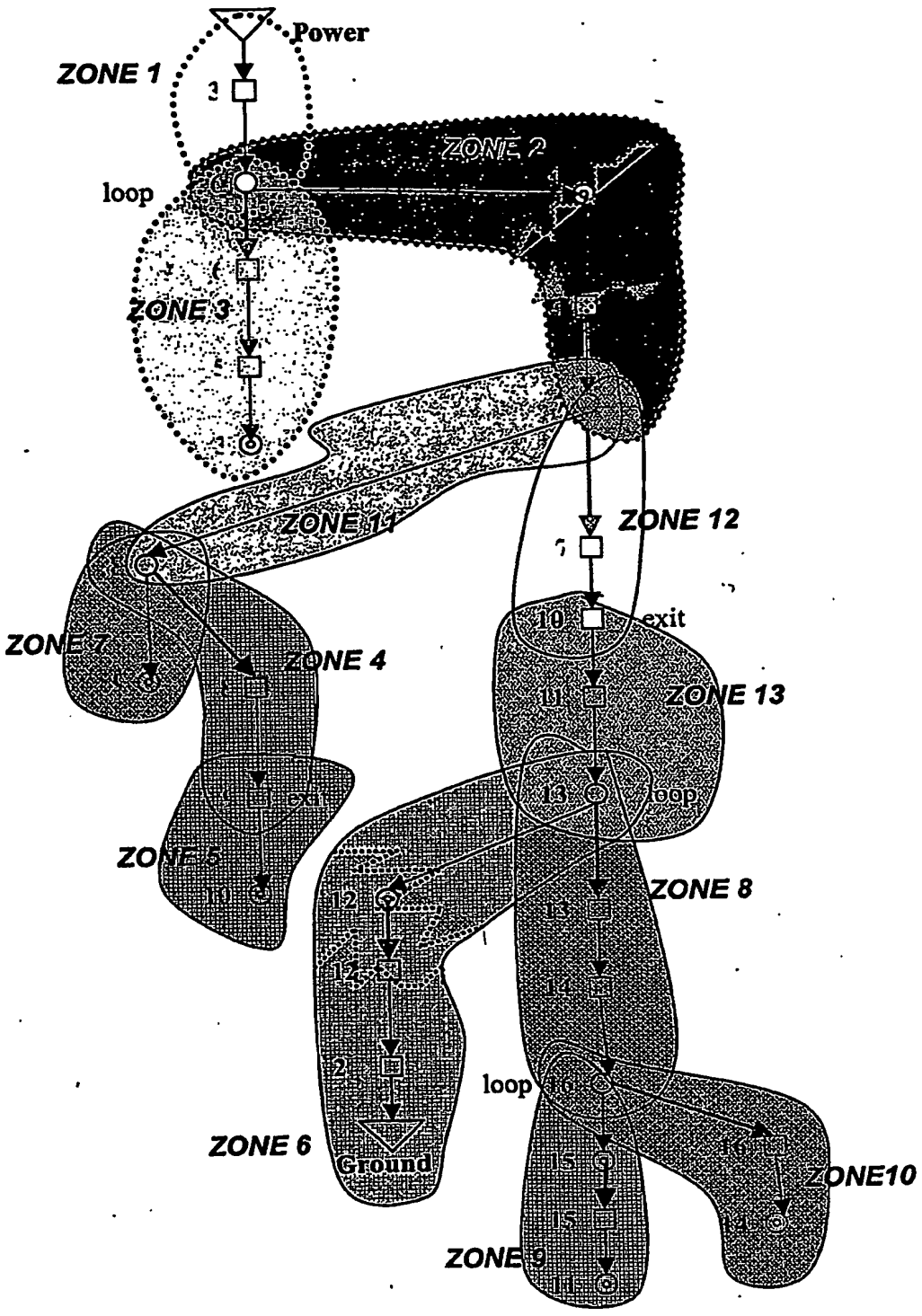


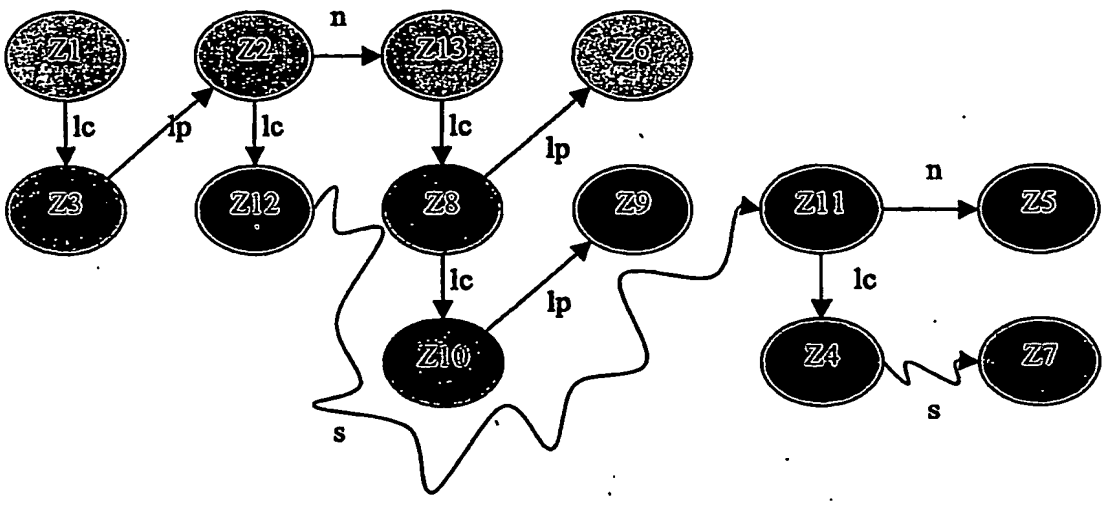
Figure 4: Zones in the Modified Structure

The Pseudo code for the above process is given below:

```
Global variables: pop flag = 0, tree_empty = 0;
Zonise (node) /* input into the function is the current node, a starting node */
{
        while (tree_empty == 0) /* this loop goes on node by node in the tree till all node
                                   have been scanned */
        {
        if (node → type = circle)
        {
                if (pop_flag != set) /* pop flag is set when a pop operation is done */
                {
                        /* an entry here means that the circle was encountered for the first
                           time */
                        /* so set the node→ visited flag */
                        /* close the zone */
                        /* since u r entering a virgin circle, u cant create the new zone as a
                           sibling to the one u just closed */
                        /* if the zone u just closed, has a valid Anchor Point and if its of
                           type Loop and if its visited flag is set, then u cannot create a
                           child zone */
                        /* accordingly create a new zone */
                        /* set child as current zone*/
                        /* push this zone and the node into the queue */
                        /* take the taken path for the node, i.e node = node→ taken */
                }
                if (pop_flag = set)
                {
                        /* an entry here means, that we r visiting a node and its associated
                           zone, that have just been popped out form the queue, hence
                           revisiting an old node */
                        /* since this node has its visited flag as set, change that flag value
                           to -1, so as to avoid any erroneous visit in the future */

                        /* if node is of type Non Loop, then spawn a new sibling zone */
                        /* if node is of type Loop, then spawn new zone as laterparent zone
                           and mark zone type as loop*/
                        /* choose the not taken path for the node */
                }
        }

        else if (node→ type = exit square)
        {
                /* close the zone */
                /* if the closed zone has a parent, i.e zone→ parent pointer is not NULL,
                   then create a new zone with link to the parent zone as type next zone */
                /* if the closed zone does not have a parent, then spawn a new zone that is
                   next to the closed zone */
```

```
            /* choose the not taken path for the node */
        }

        else if (node→ type is dot and node→ taken = NULL)
        {
            /* close zone */
            /* choose node to be considered next by popping out from the queue */
            /* in case the queue is empty, all nodes in tree have been scanned */
            /* set pop flag */
        }

        else if (node→ type = dot and node→ taken != NULL)
        {
            /* this is just a break in the current zone */
            /* create temp stop1 and tempstart1 pointers*/
            /* choose node→ taken path */
        }
        }/* end of the first while loop */
}
```

Once the zones have been identified in the structure, certain relationships can be observed among them. These form the basis of extraction of parallelism at the level of zones. A zone inside a control structure is the 'later child' of the zone outside the structure. Hence the zone outside a control structure and occurring before (in code sequence) the zone inside a control structure is a 'former parent' of the zone present inside. But, the zone outside a control structure and occurring after (in code sequence) the zone inside the structure is referred to as the 'later parent'. Similarly the child in this case would be a 'former child'. A zone occurring after another zone and not related through a control structure is the 'next' of the earlier one. After parsing through the structure thru the zonal relationship as shown in Figure 5 is obtained.



S: sibling relationship
LC: later child relationship
Lp: later parent relationship
In all types, destination zone is (lc/s/lp) of source zone
The shaded zones are Loop types.

Figure 5: Initial Zone Structure obtained

This is referred to as the 'initial zone structure'. The term initial, is used because, some links need to be created and some existing ones, need to be removed. This process is explained in the section below.

Step 6: Further Modification of the 'initial zone structure'
Some of the relationships that were discussed in the previous step cannot exist with the existing set of links and others are redundant. For example in in Figure5, we see that Z1 can be connect to Z2 thru 'n'
Z12 can be connected to Z13 thru 'lp'
Z13 can be connected to Z6 thru 'n'
Z8 can be connected to Z9 thru 'n'
Z4 can be connected to Z5 thru 'lp'
Z5 can be connected to Z13 thru 'lp'
Z7 can be connected to Z5 thru 'lp'

But Z8's relationship to Z6 thru 'lp' is false, coz no node can have both 'n' and 'lp' links.
In such a case, the 'lp' link should be removed.
Therefore some rules need to be followed to establish 'n' and 'lp' type links, if they don't exist.
To form an 'n' link:

If a zone (1) has an 'lc' link to zone (2), and if that zone (2) has a 'lp' link to a zone (3), then an 'n' link can be established between 1 and 3. This means that if zone (1) is of type 'loop', then zone (3) will now be classified as type 'loop' also.

To form an 'lp' type links if it doesn't exist:
If a zone (1) has an 'fp' link to zone (2), and if that zone (2) has an 'n' link to a zone (3), then an 'lp' link can be established between 1 and 3
If a zone (1) has an 'lp' link to zone (2), and also has an 'n' link to zone (3), then first, remove the 'lp' link 'to zone (2)' from zone (1) and then, place an 'lp' link from zone (3) to zone (2).
This provides the 'comprehensive zone structure' as shown in Figure 6 (with cancelled links) and in Figure 7 (with all cancelled links removed).

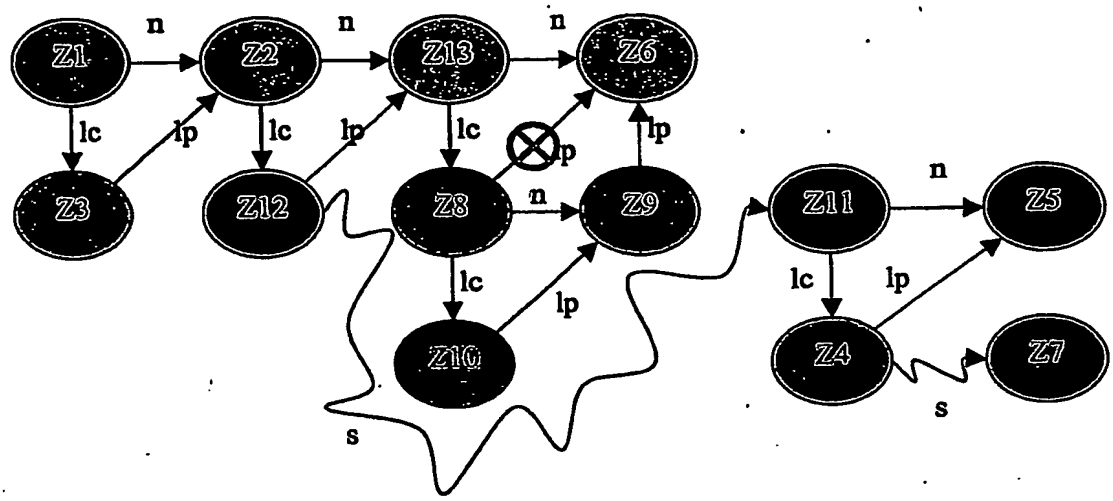


Figure 6: Comprehensive Zone structure with cancelled links shown

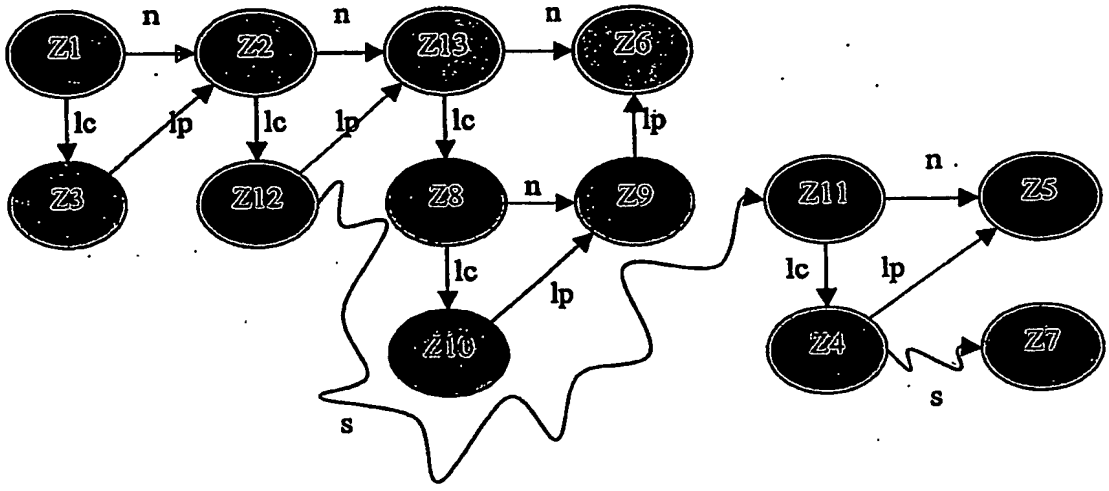


Figure 7: Comprehensive Zone structure with cancelled links removed

To identify parallelism and hence compulsorily sequential paths of execution, the following approach is adopted. Firstly, the comprehensive zone structure obtained, is ordered sequentially by starting at the first zone and traversing along an 'lc – lp' path. If a Sibling link is encountered it is given a parallel path. The resulting structure is shown in Figure 8.

THIS PAGE INTENTIONALLY LEFT BLANK.

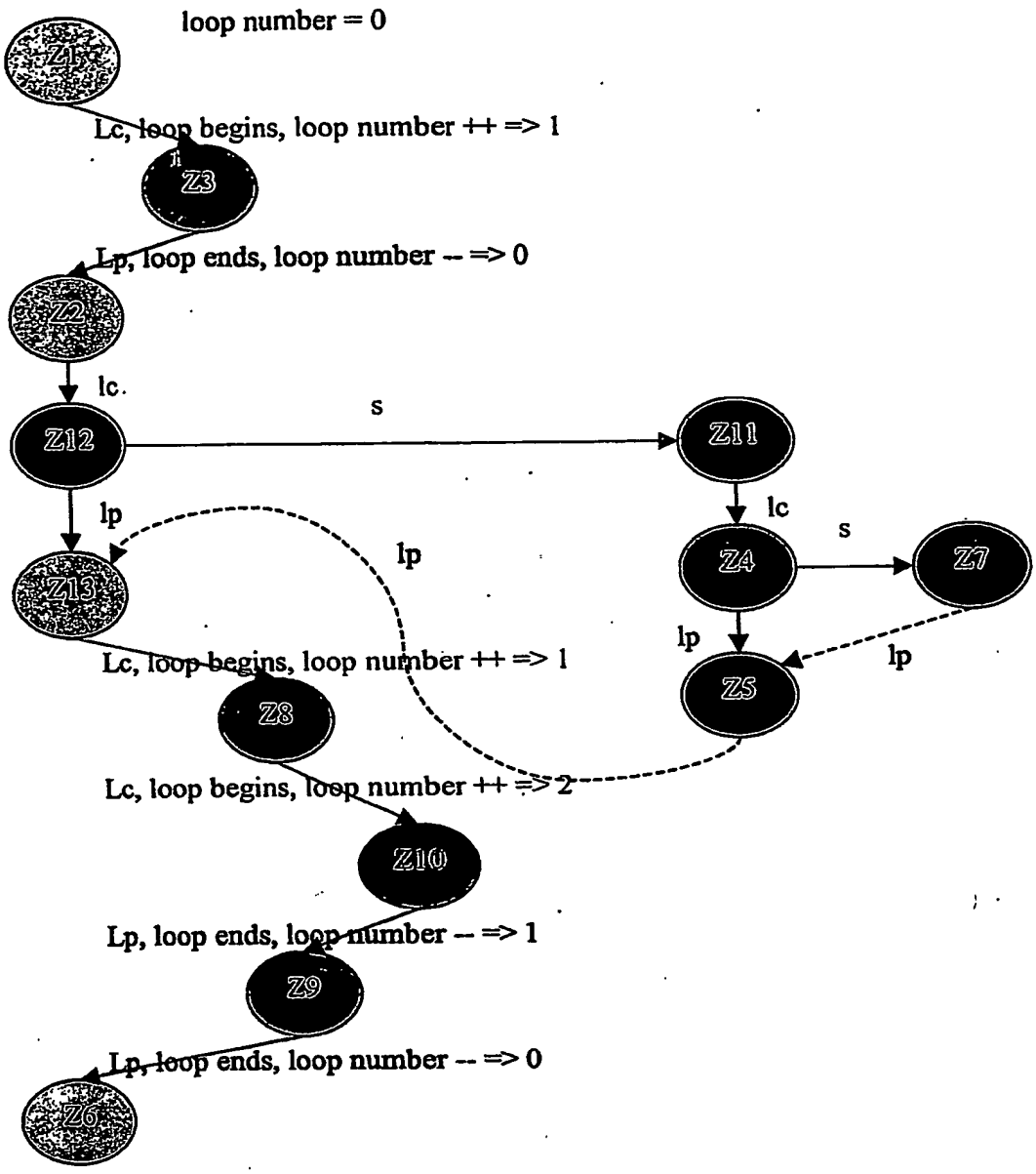


Figure 8: Sequentially Ordered Zones

To establish parallelism between a zone (1) of loop count A and its upper zone (2) of loop count B, where A < B, check for data dependency between zone 1 and all zones above it upto and including the zone with the same loop count as zone 2.
In the example above, to establish parallelism b/w zone 6 and zone 9, check for dependencies b/w zone 6 and 9, 10, 8. If there is no dependency then zone 6 is parallel to zone 8.

To establish parallelism between a zone (1) of loop count A and its upper zone (2) of loop count B, where A = B, direct dependency check needs to be performed.
To establish parallelism between a zone (1) of loop count A and its upper zone (2) of loop count B, where A > B, direct dependency check needs to be performed. Then, the zone (1) will now have to have an iteration count of (its own iteration count zone (2)'s iteration count).
When a zone rises like a bubble and is parallel with another zone in the primary path, and reaches a dependency, it is placed in a secondary path. No bubble in the secondary path is subjected to dependency testing.
After a bubble has reached its highest potential, and stays put in a place in the secondary path, the lowest bubble in the primary path is checked for dependency on its upper fellow.
If the upper bubble happens to have a different loop count number, then as described earlier, testing is carried out. In case a parallelism cannot be obtained, then this bubble, is clubbed with the set of bubbles ranging from its upper fellow, till and inclusive of the bubble up the chain with the same loop count as its upper fellow. A global i/o parameter set is created for this new coalition. Now this coalition will attempt to find dependencies with its upper fellow. The loop count for this coalition will be bounding zone's loop count. Any increase in the iteration count of this coalition will reflect on all zones inside it. In case a bubble wants to rise above another one which has a sibling/ reverse sibling link, there will be speculative parallelism.
The algorithm should start at multiple points, one by one. These points can be obtained by starting from the top zone and traversing down, till a sibling split is reached. Then this zone should be remembered, and one of the paths taken. This procedure is similar to the stack saving scheme used earlier in the zonise function.
Another Pre-processing step is used that loop unrolls every iterative segment of a CDFG that does not have conditional branch instructions inside it and whose iterative count is known at compile time.

9 Appendix B

```
#include<stdio.h>

void main()
{
  int i,j,k,l;


  i = 10;
  j = 1* 4;

  if ( j > 5 )
  {
    k=findsum(i,j);
    l = 4+k;
  }
  else
  {
    k = findsum(i,j);
    l = k*10;
  }
```

```
}


int findsum(int a,int b)
{
  int i,j,k;

 k=4;
 for(i=0;i<10;i++)
  k = k + 1;

 j = findsub(k,a);

 return j;
}


int findsub(int x,int y)
{
   int t;

   t = x-y;

    return(t);
}
```

10 Appendix C

Main.s

```
    .file "main.c"
gcc2_compiled.:
.section ".text"
    .align 4
    .global main
    .type  main,#function
    .proc 020
main:
    !#PROLOGUE# 0
    save  %sp, -128, %sp
    !#PROLOGUE# 1
    mov   10, %o0
    st    %o0, [%fp-20]
    mov   4, %o0
    st    %o0, [%fp-24]
    ld    [%fp-24], %o0
    cmp   %o0, 5
    ble   .LL3
    nop
    ld    [%fp-20], %o0
    ld    [%fp-24], %o1
    call  findsum, 0
     nop
    st    %o0, [%fp-28]
    ld    [%fp-28], %o0
```

```
   add   %o0, 4, %o1
   st    %o1, [%fp-32]
   b     .LL4
    nop
.LL3:
   ld    [%fp-20], %o0
   ld    [%fp-24], %o1
   call  findsum, 0
    nop
   st    %o0, [%fp-28]
   ld    [%fp-28], %o0
   mov   %o0, %o2
   sll   %o2, 2, %o1
   add   %o1, %o0, %o1
   sll   %o1, 1, %o0
   st    %o0, [%fp-32]
.LL4:
.LL2:
   ret
   restore
.LLfe1:
   .size  main,.LLfe1-main
   .ident      "GCC: (GNU) 2.95.2 19991024 (release)"
```

Findsum.s

```
   .file "findsum.c"
gcc2_compiled.:
.section ".text"
   .align 4
   .global findsum
   .type  findsum,#function
   .proc 04
findsum:
   !#PROLOGUE# 0
   save  %sp, -128, %sp
   !#PROLOGUE# 1
   st    %i0, [%fp+68]
   st    %i1, [%fp+72]
   mov   4, %o0
   st    %o0, [%fp-28]
   st    %g0, [%fp-20]
.LL3:
   ld    [%fp-20], %o0
   cmp   %o0, 9
   ble   .LL6
   nop
   b     .LL4
    nop
.LL6:
   ld    [%fp-28], %o0
   add   %o0, 1, %o1
   st    %o1, [%fp-28]
.LL5:
   ld    [%fp-20], %o0
   add   %o0, 1, %o1
   st    %o1, [%fp-20]
   b     .LL3
    nop
```

```
.LL4:
   ld    [%fp-28], %o0
   ld    [%fp+68], %o1
   call  findsub, 0
    nop
   st    %o0, [%fp-24]
   ld    [%fp-24], %o0
   mov   %o0, %i0
   b     .LL2
    nop
.LL2:
   ret
   restore
.LLfe1:
   .size  findsum,.LLfe1-findsum
   .ident       "GCC: (GNU) 2.95.2 19991024 (release)"
```

Findsub.s

```
   .file "findsub.c"
gcc2_compiled.:
.section ".text"
   .align 4
   .global findsub
   .type  findsub,#function
   .proc 04
findsub:
   !#PROLOGUE# 0
   save  %sp, -120, %sp
   !#PROLOGUE# 1
   st    %i0, [%fp+68]
   st    %i1, [%fp+72]
   ld    [%fp+68], %o0
   ld    [%fp+72], %o1
   sub   %o0, %o1, %o0
   st    %o0, [%fp-20]
   ld    [%fp-20], %o0
   mov   %o0, %i0
   b     .LL2
    nop
.LL2:
   ret
   restore
.LLfe1:
   .size  findsub,.LLfe1-findsub
   .ident       "GCC: (GNU) 2.95.2 19991024 (release)"
```

11 Appendix D

Expanded main function

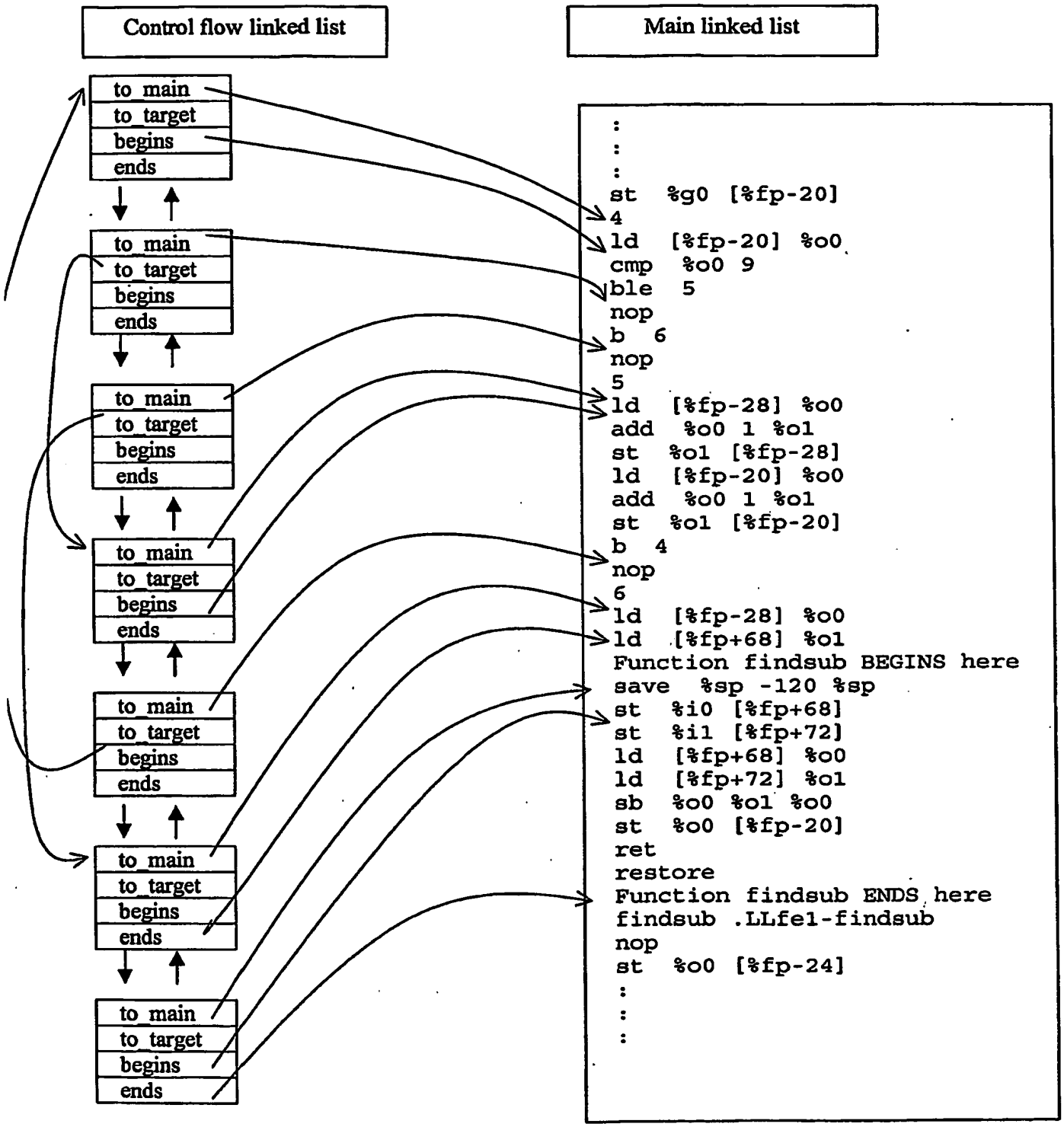
```
Function main BEGINS here

 save  %sp -128 %sp
```

```
mov  10 %o0
st  %o0 [%fp-20]
mov  4 %o0
st  %o0 [%fp-24]
ld  [%fp-24] %o0
cmp  %o0 5
ble  0
nop
ld  [%fp-20] %o0
ld  [%fp-24] %o1
Function findsum BEGINS here

save  %sp -128 %sp
st  %i0 [%fp+68]
st  %i1 [%fp+72]
mov  4 %o0
st  %o0 [%fp-28]
st  %g0 [%fp-20]
4
ld  [%fp-20] %o0
cmp  %o0 9
ble  5
nop
b  6
nop
5
ld  [%fp-28] %o0
add  %o0 1 %o1
st  %o1 [%fp-28]
7
ld  [%fp-20] %o0
add  %o0 1 %o1
st  %o1 [%fp-20]
b  4
nop
6
ld  [%fp-28] %o0
ld  [%fp+68] %o1
Function findsub BEGINS here

save  %sp -120 %sp
st  %i0 [%fp+68]
st  %i1 [%fp+72]
ld  [%fp+68] %o0
ld  [%fp+72] %o1
sb  %o0 %o1 %o0
st  %o0 [%fp-20]
ld  [%fp-20] %o0
mov  %o0 %i0
b  10
nop
10
ret
restore
11
Function findsub ENDS here
findsub .LLfe1-findsub
nop
st  %o0 [%fp-24]
ld  [%fp-24] %o0
mov  %o0 %i0
```

```
b  8
nop
8
ret
restore
9
Function findsum ENDS here
findsum .LLfe1-findsum
nop
st  %o0 [%fp-28]
ld  [%fp-28] %o0
add  %o0 4 %o1
st  %o1 [%fp-32]
b  1
nop
0
ld  [%fp-20] %o0
ld  [%fp-24] %o1
Function findsum BEGINS here

save  %sp -128 %sp
st  %i0 [%fp+68]
st  %i1 [%fp+72]
mov  4 %o0
st  %o0 [%fp-28]
st  %g0 [%fp-20]
4
ld  [%fp-20] %o0
cmp  %o0 9
ble  5
nop
b  6
nop
5
ld  [%fp-28] %o0
add  %o0 1 %o1
st  %o1 [%fp-28]
7
ld  [%fp-20] %o0
add  %o0 1 %o1
st  %o1 [%fp-20]
b  4
nop
6
ld  [%fp-28] %o0
ld  [%fp+68] %o1
Function findsub BEGINS here

save  %sp -120 %sp
st  %i0 [%fp+68]
st  %i1 [%fp+72]
ld  [%fp+68] %o0
ld  [%fp+72] %o1
sb  %o0 %o1 %o0
st  %o0 [%fp-20]
ld  [%fp-20] %o0
mov  %o0 %i0
b  10
nop
10
ret
```

```
restore
11
Function findsub ENDS here
findsub .LLfe1-findsub
nop
st  %o0 [%fp-24]
ld  [%fp-24] %o0
mov  %o0 %i0
b  8
nop
8
ret
restore
9
Function findsum ENDS here
findsum .LLfe1-findsum
nop
st  %o0 [%fp-28]
ld  [%fp-28] %o0
mov  %o0 %o2
sll  %o2 2 %o1
add  %o1 %o0 %o1
sll  %o1 1 %o0
st  %o0 [%fp-32]
1
2
ret
restore
3
Function main ENDS here
```

12 Appendix E



13 Appendix F

In this section the pseudo ANSI C codes for the test-bench algorithms are presented.

Note: For an indepth-analysis and explanation on all graphics algorithms, please refer to the book: "Computer Graphics: Principles and Practise" Second edition in C, by Foley, van Dam, Feiner and Hughes.

Cohen Sutherland Line Clipping

```
typedef unsigned int outcode;
enum {TOP=0x1, BOTTOM=0x2, RIGHT=0x4, LEFT=0x8};

void CohenSutherlandLineClipAndDraw (
        double x0, double y0, double x1, double y1, double xmin, double xmax,
        double ymin, double ymax, int value)
/* Cohen-sutherland clipping algorithm for line P0 = (x0,y0) to P1 = (x1,y1) and */
/* clip rectangle with diagonal from (xmin,ymin) to (xmax,ymax) */
{
        /* Outcodes for P0, P1 and whatever point lies outside the clip rectangle */
        outcode outcode0, outcode1, outcodeOut;
        boolean aacept = FALSE, done = FALSE;
        outcode0 = CompOutCode (x0,y0,xmin,xmax,ymin,ymax);
        outcode1 = CompOutCode (x1,y1,xmin,xmax,ymin,ymax);
        do {
                if (!(outcode0 | outcode1)) {
                        accept = TRUE; done = TRUE;
                } else if (outcode0 & outcode1)
                        done = TRUE;
                else {
                        double x,y;
                        outcodeOut = outcode0?outcode0:outcode1;
                        if (outcodeOut & TOP) {
                                x = x0 + (x1 - x0)*(ymax - y0) / (y1 - y0);
                                y = ymax;
                        } else if (outcodeOut & BOTTOM) {
                                x = x0 + (x1- x0)*(ymin - y0) / (y1 - y0);
                                y = ymin;
                        } else if (outcodeOut & RIGHT) {
                                y = y0 + (y1- y0)*(xmax - x0) / (x1 - x0);
                                x = xmax;
                        } else {
                                y = y0 + (y1- y0)*(xmin - x0) / (x1 - x0);
                                x = xmin;
                        }

                        if (outcodeOut == outcode0) {
```

```
                        x0 = x; y0 = y; outcode0 = CompOutCode
(x0,y0,xmin,xmax,ymin,ymax);
                    } else {
                        x1 = x; y1 = y; outcode1 = CompOutCode
(x1,y1,xmin,xmax,ymin,ymax);
                    }
                }
        } while (done == FALSE);

        if(accept)
                MidpointLineReal (x0,yo,x1,y1,value);
}

outcode CompOutode (
        double x, double y, double xmin, double xmax, double ymin, double ymax)
{
        outcode code = 0;
        if (y<ymax)
                code |= TOP;
        else if (y<ymin)
                code |= BOTTOM;
        if (x>xmax)
                code |= RIGHT;
        else if (x<xmin)
                code |= LEFT;
        return code;
}

void MidpointLineReal (double x0,double yo,double x1,double y1,double value)
{
        double dx = x1 - x0;
        double dy = y1 - y0;
        double d = 2*dy - dx;
        double incrE = 2*dy;
        double incrNE = 2*(dy - dx);
        double x = x0;
        double y = y0;
        WritePixel (x,y,value);

        while (x<x1) {
                if (d<=0) {
                        d += incrE;
                        x++;
                } else {
                        d += incrNE;
                        x++;
                        y++;
```

```
            }
            WritePixel (x,y,value);
        }
}
```

Mid-point Ellipse Scan Conversion

```
void MidpointEllipse (int a, int b, int value)
/* Assumes center of ellipse is at the origin. Note that overflow may occur */
/* for 16-bit integers because oft the squares */
{
        double d2;
        int x=0;
        int y = b;
        double d1 = b^2 - (a^2b) + (0.25a^2);
        EllipsePoints (x,y,value); /* The 4-way symmetrical WritePixel */

        while (a^2(y - 0.5) > b^2(x + 1)) {
                if (d1 < 0)
                        d1 += b^2(2x + 3);
                else {
                        d1 += b^2(2x + 3) + a^2(-2y + 2);
                        y--;
                }
                x++;
                EllipsePoints(x,y,value);
        }

        d2 = b^2(x + 0.5)^2 + a^2(y - 1)^2 - a^2b^2;
        while (y > 0) {
                if (d2 < 0) {
                        d2 += b^2(2x + 2) + a^2(-2y + 3);
                        x++;
                } else
                        d2 += a^2(-2y + 3);
                y--;
                EllipsePoints(x,y,value);
        }
}
```

The bitBlock Transfer Algorithm

```
typedef struct {
        point topLeft, bottomRight;
}rectangle;

typedef struct {
        cha *base;
```

```
        int width;
        rectangle rect;
} bitmap;

typedef struct {
        unsigned int bits:32;
} texture;

typedef struct {
        char *worldptr;
        int bit;
} bitPointer;

void bitBlt(
        bitmap map1;
        point point1;
        texture tex;
        bitmap map2;
        rectangle rect2;
        writeMode mode)
{
        int width;
        int height;
        bitPointer p1,p2;

        clip x_values;
        clip y-values;

        width = rect2.bottomRight.x - rect2.topLeft.x;
        height = rect2.bottomRight.y - rect2.topLeft.y;

        if (width < 0 || height < 0)
                return;

        p1.wordptr = map1.base;
        p1.bit = map1.rect.topLeft.x % 32;

        /* And the first bin in the bitmap is a few bits further in */
        /* Increment p1 unitl it points to the specified point in the first bitmap */
        IncrementPointer (p1,point1.x - map1.rect.topLeft.x + map1.width *
                                        (point1.y - map1.rect.topLeft.y));

        /* Same for p2 - it points to the origin of the destination rectangle */
        p2.worldptr = map2.base;
        p2.bit = map2.rect.topLeft.x % 32;
        IncrementPointer (p2,rect2.topLeft.x - map2.rect.topLeft.x +
                                        map2.widrh * (rect2.topLeft.y - map2.rect.topLeft.y));
```

```
    if(p1 < p2) {
        /* The pointer p1 comes before p2 in memory; if they are in the same bitmap */
        /* the origin of the source rectangle is either above the origin for the */
        /* above destaination or, if at the same level, to the left of it */

        IncrementPointer (p1, height * map1.width + width);
        /* Now p1 points to the lower right word of the rectangle */
        IncrementPointer (p2, height * map1.width + width);
        /* Same for p2, but the destination rectangle */
        point1.x += width;
        point1.y += height;
        /* Thios point is now just beyond the lower right in the rectangle */
        while (height-- > 0){
            /* Copy rows from the source to the target bottom to top, right to left */
            DecrementPointer (p1, map1.width);
            DecrementPointer (p2, map2.width);
            temp_y = point1.y % 32; /* used to index into texture */
            temp_x = point1.x % 32;
            /* Now do the real bitBlt from bottom right to top left */
            RowBltNegative (p1, p2, width, BitRotate(tex[temp_y],temp_x), mode);
        } /* while */
    } else { /* if p1 >= p2 */
        while (height-- > 0) {
            /* Copy rows fro source to destaination, top to bottom, left to right */
            /* Do the real bitBlt, from topleft tpo bottom right */
            RowBltPositive (same arguments as before);
            increment pointers;
        } /* while */
    } /* else */
} /* bitBlt */

void Clip Values (bitmap *map1, bitmap *map2, point *point1, rectangle *rect2)
{
    if (*point1 not inside *map1){
        adjust *point1 to be inside *map1;
        adjust origin of *rect2 by the same amount;
    }
    if (originof *rect2 not inside *map2){
        adjust origin of *rect2 to be inside *map2;
        adjust *point1 by the same amount;
    }
    if (opposite corner of *rect2 not inside *map2)
        adjust opposite corner of *rect2 to be inside;
    if (opposite corner of corresponding rectangle in *map1 not insode *map1)
        adjust opposite corner of rectangle;
} /*ClipValues */
```

```
void RowBltPositive(
        bitPtr p1, bitPtr p2;   /* Source and destination pointers */
        int n;                  /* How many bits to copy */
        char tword;             /* Texture word */
        writeMode mode)         /* Mode to blt pixels */
{
    /* Copy n bits from position p1 to position p2 according to the mode */
    while (n-- > 0) {
        if (BitIsSet (tword,32))/* If texture says it is OK to copy..*/
            MoveBit (p1,p2,mode); /* then copy the bit */
        IncrementPointer (p1);
        IncrementPointer (p2);
        RotateLeft (tword);         /* Rotate bits in tword to the left */
    } /* while */
} /RowBltPositive */
```

Phong Shading

```
double db1=2.5,db2=65535.,pi;
int colors[]={3,6,10,13,6,3,10,13,6,3,13,10},
    d[]={640,350,1},
    i,k,
    palette[]={000,010,001,011,020,002,022,077,
               040,004,044,060,006,066,007,077},
    x,y,x_min,x_max,y_min,y_max;
int min,sec;
unsigned short random;

main()

{
  double a,b,c,l0,l1,l2,ln,ln1,n0,n1,n2,p,q,r=128,s,t,v[12][3];
  int n;


  int graphdriver = DETECT, graphmode;
  int color;

  initgraph(&graphdriver, &graphmode, "");

/*  for (n=0;n<16;n++)  */


#ifdef Intel
  printf("\n\t\t        80387 Phong Shading Demonstration Program\n");
#else
  printf("\n\t\t\t    Phong Shading Demonstration\n");
#endif
/*  printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n \n\n\n\n\n\n\n");
  start=clock(); */


 /* Pixel aspect ratio. Original value is 1.3 whic works with EGA*/
```

```
/* This is hence the version for my - ThL - EIZO VGA Card        */
 a=1.0;
 /* Screen center coordinates */
 b=0.5*(d[0]-1);     /* x-position */
 c=0.5*(d[1]-1);     /* y-position */
 /* Unit length light source vector */
 l0=-1/sqrt(3.);
 l1=l0;
 l2=-l0;
 /* Ratio circumference to diameter of a circle */
 pi=4*atan(1.);
 /* A dozen vertices evenly spread over a unit sphere */
 v[0][0]=0;
 v[0][1]=0;
 v[0][2]=1;
 s=sqrt(5.);
 for (i=1;i<11;i++)  {
   p=pi*i/5;
   v[i][0]=2*cos(p)/s;
   v[i][1]=2*sin(p)/s;
   v[i][2]=(1.-i%2*2)/s;
 }
 v[11][0]=0;
 v[11][1]=0;
 v[11][2]=-1;

 /* Loop to Phong shade each pixel */
 y_max=c+r;
 y_min=2*c-y_max;
 for (y=y_min;y<=y_max;y++) {
    s=y-c;
    n1=s/r;
    ln1=l1*n1;
    s=r*r-s*s;
    x_max=b+a*sqrt(s);
    x_min=2*b-x_max;
    for (x=x_min;x<=x_max;x++)  {
      t=(x-b)/a;
      n0=t/r;
      t=sqrt(s-t*t);
      n2=t/r;
      /* Compute dot product and clamp to positive value */
      ln=l0*n0+ln1+l2*n2;
      if (ln<0) ln=0;
      /* cos(e.r)**27 */
      t=ln*n2;
      t+=t-l2;
      t*=t*t;
      t*=t*t;
      t*=t*t;
      /* Nearest vertex to normal yields max dot product */
      /* Get its color */
      for (i=0,p=0;i<11;i++)
        if (p<(q=n0*v[i][0]+n1*v[i][1]+n2*v[i][2]))  {
           p=q;
           k=colors[i];
        }/*end for*/
      /* Aggregate ambient, diffuse, and spectacular intensities
         do dither */
      random=37*random+1;
      i=k-db1+db1*ln+t+random/db2;
```

```
      /* Clamp values outside range of three color level to black or white */
        if (i < (k-2)) i=0;
      else
      if (i > k) i=15;
      putpixel(x,y,i);
      }/*end for*/
    }/*end for*/

exit:
    delay(5000);
    closegraph();

    }/*end main*/
```

14 Appendix G

Algorithm:

Task schedule (G(V,E), CTRL_VARS[N], PE = {PE1,PE2.....PEM})
For each combination of CTRL_VARS do
{
Generate a DFG Gsub(V,E,CTRL_VARS[I]) which is a sub-graph of G(V,E). Only the nodes and edges in the control flow corresponding to the current combination of CTRL_VARS are included in this sub-graph.
Generate the PCP schedule of Gi. Let the schedule be PCP_sched[I] and the delay be PCP_delay[I].
}
Sort PCP_sched and PCP_delay and Gsub in decreasing order of PCP_delay[I].

Generate the Branch and bound schedule for Gsub[0], the sub-graph with the worst PCP_delay. Let the schedule be BB_sched[I=0] and the delay be BB_delay[I=0].
Initialize worst_bb_delay = BB_delay[0]

For all the other sub-graphs do
{
if (PCP_delay[I] < worst_bb_delay) then
BB_sched[I] = PCP_sched[I];
BB_delay[I] = PCP_delay[I];
else
Generate BB_sched[I] and BB_delay[I];
If (BB_delay[I] > worst_bb_delay[I]) then
Worst_bb_delay = BB_delay[I];
}

Generate the branching tree with the help of the G(V,E). In this branching tree, the edge represents the choices (K and K') and the node represents the variable (K)
Initialize the current path to the one leading from the top to the leaf in such a way that the DFG corresponding to this path gives the worst_bb_delay. The path is nothing but a list of edges tracing from the top node till the leaf.

A Methodology to Design a Dynamically Reconfigurable Media Processor

Aravind R. Dasu
and
S. Panchanathan *Fellow, IEEE*

MASES'03
October 29, 2003

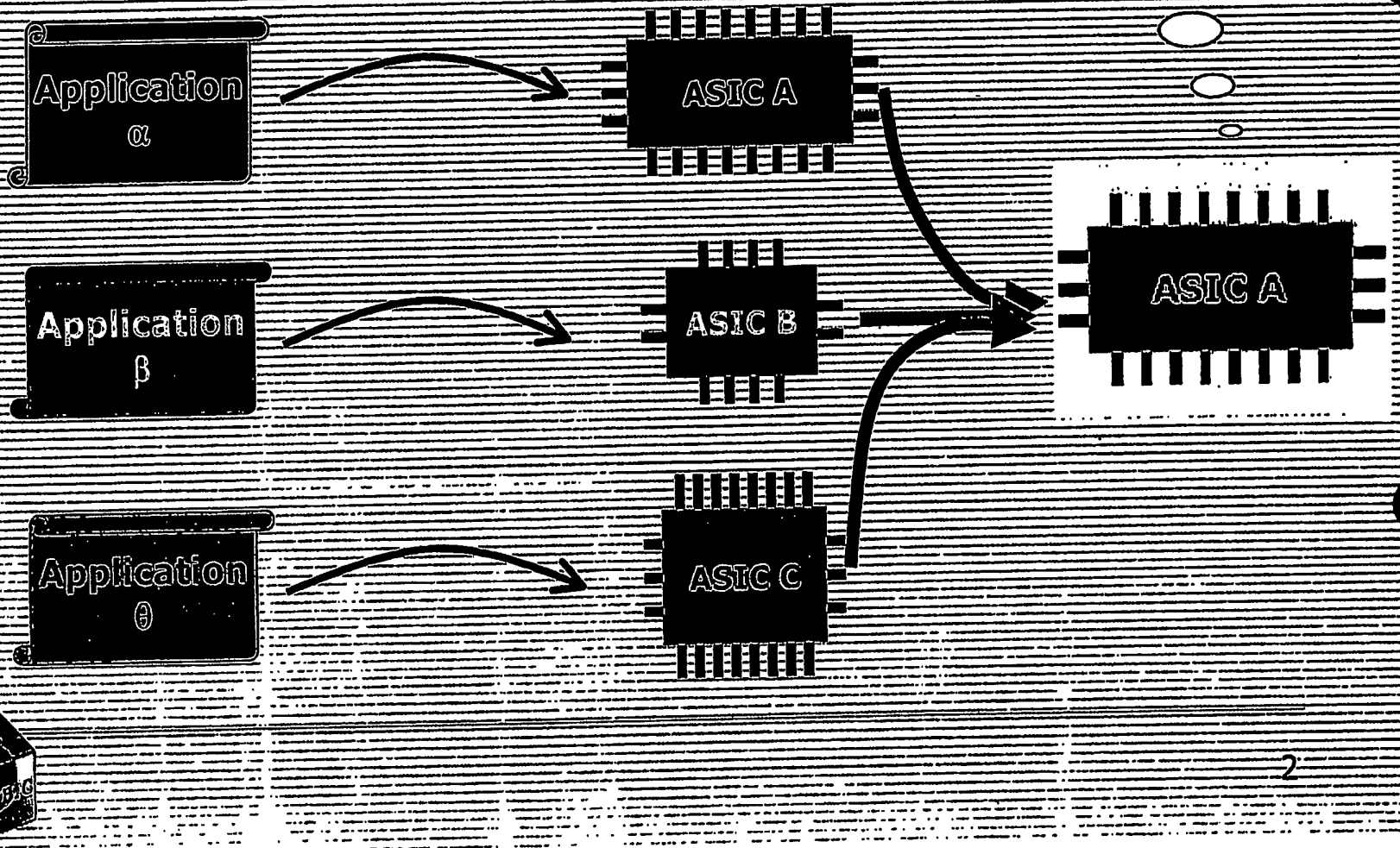
ASU

Reconfiguration

What to do ?
(Compute Intensive Applications)

How best to do (individually) ?

Circuit adaptability



What is the Target ?

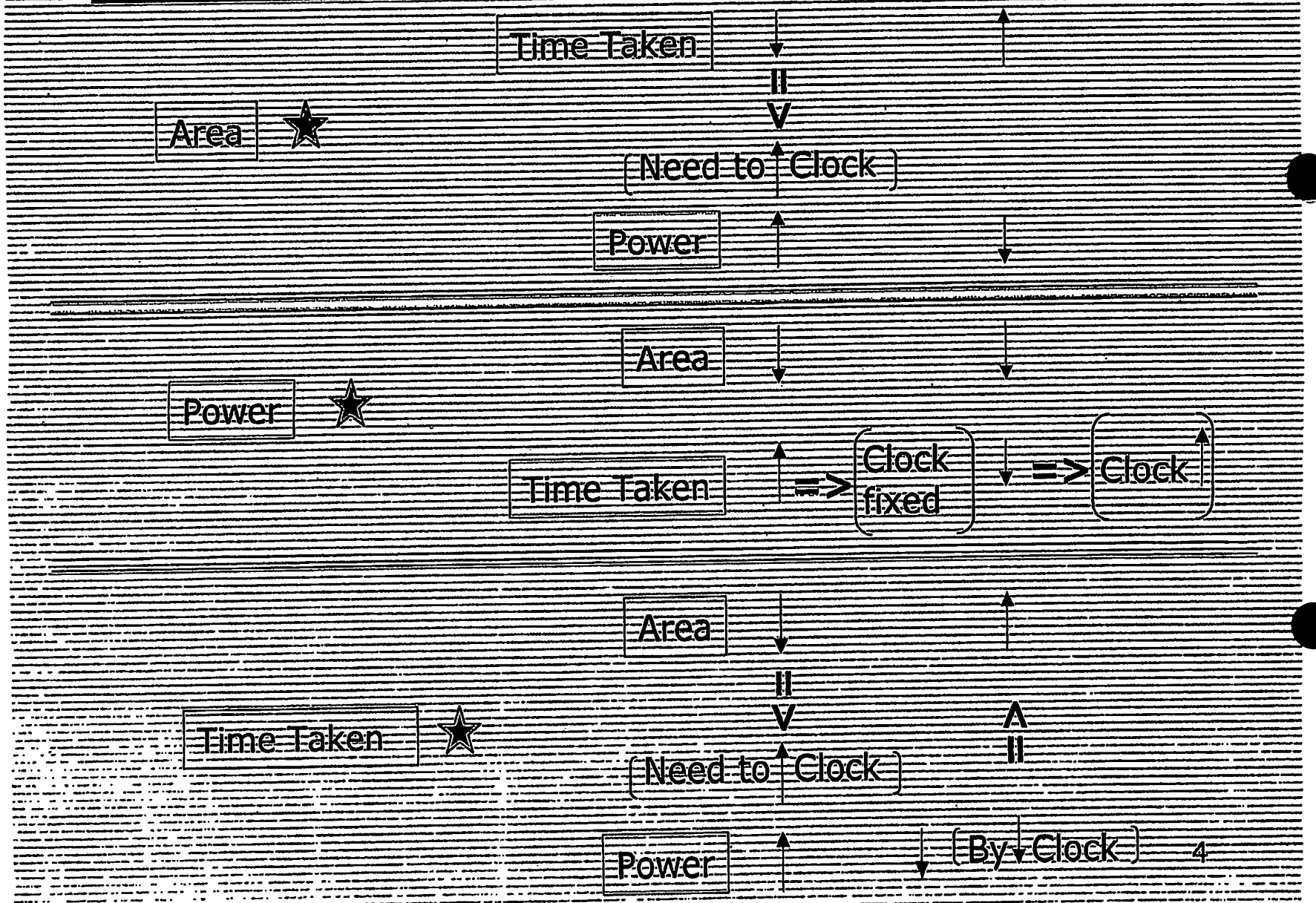
Design a low area, low power consuming and fast reconfigurable processor which should provide the ability to swap applications at run time.

This can be guided through an optimization approach.

Variables involved:

- **Area (Soft and Hard Limits)**
  - **Routing / Interconnection (affected by reconfiguration)**
  - **Processing elements with associated storage (affected by clustering)**
- **Power (Soft and Hard Limits)**
- **Time (Soft and Hard Limits)**

Behavior of Variables



Breaking Down the Bigger Problem into Smaller Parts

Obtain graphs (CDFGs in IR) of application(s) through Lance compiler.

- Identify Clusters (Reconfigurable regions) from graphs

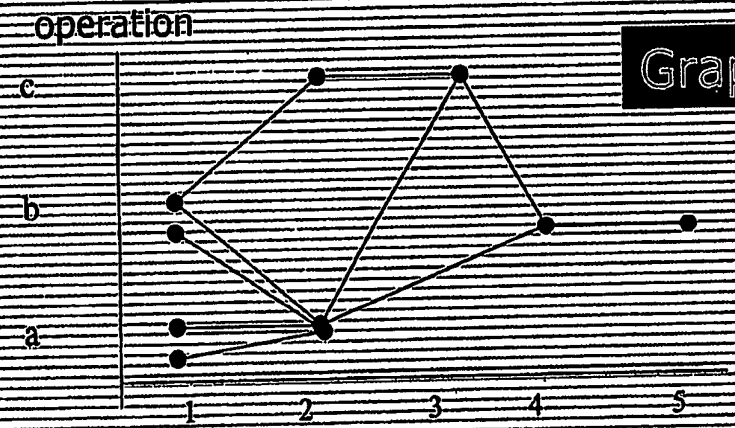
Obtain Resource count (processing units)

Schedule (map) CCDFGs on the available resources

Overview of Architecture

2. Identify Clusters (Reconfigurable Regions) from graphs

Proposed Approach



Graph

Beneficial for any future graph manipulations

Adjacency Matrix

	a2	b2	c2	a3	.....
a1	2				
b1	2		1		
c1					
a2					
.....					

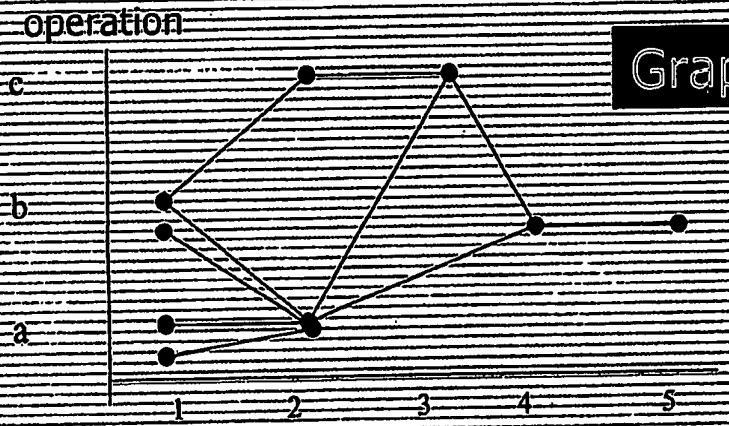
[1,...]

[1, 3,...]

a1-a2, a1-a2, b1-a2, b1-a2,
b1-c2, a2-b4, a2-c3, c2-c3,
c3-b4, b4-b5.

String

Proposed Approach

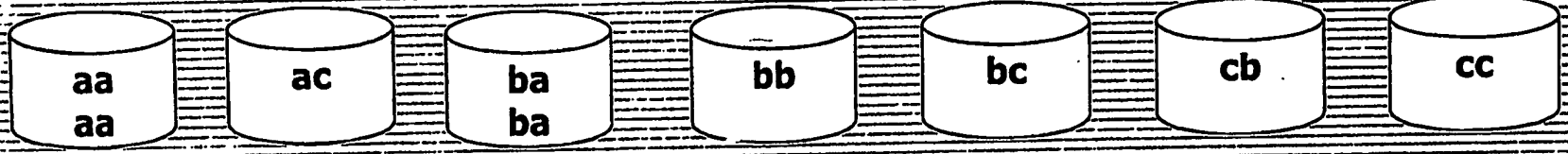


Graph

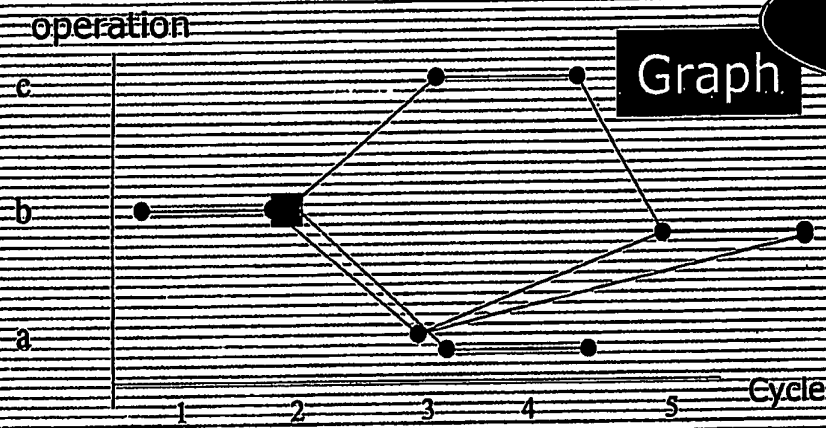
a1-a2, a1-a2, b1-a2,
b1-a2,b1-c2, a2-b4,
a2-c3, c2-c3, c3-b4,
b4-b5.

Sort
&
Hide cycle

aa, aa, ac, ba, ba, bb, bc, cb, cc



Proposed Approach

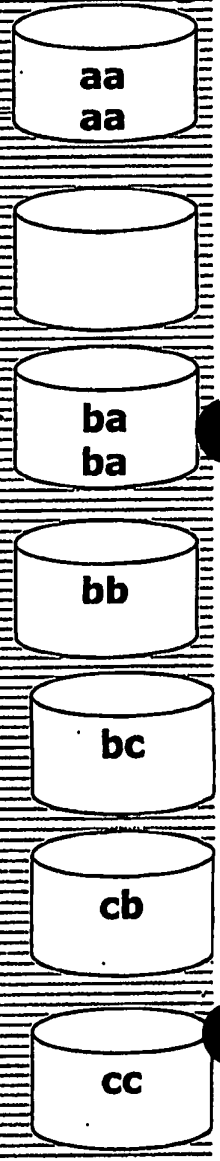


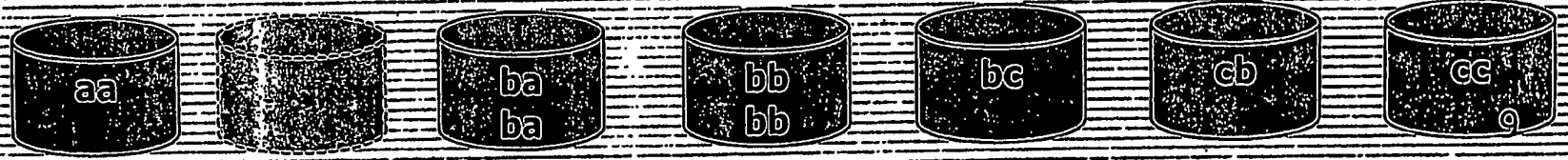
Graph

After validating ba, its previous Header was unmarked. So remove ac

Not a header in previous graph
So, eliminate

aa, ab, ab, ba, ba, bb, bb, bc, cb, cc

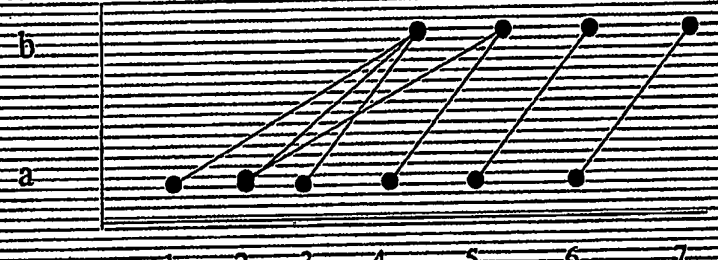


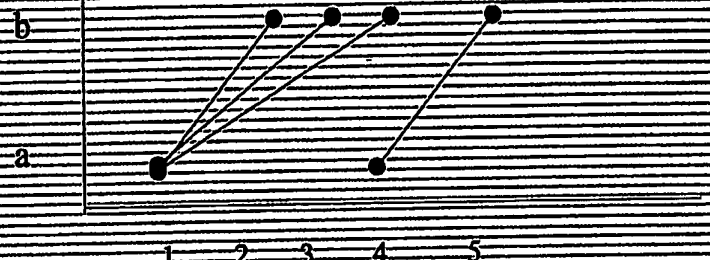


Proposed Approach

Once both bin sequences are obtained:

unmask the fan-in (of destination nodes), fan-out (of source nodes) information and collate the numbered nodes.





a1b4	a1a2a3b4
a2b4	a2a4b5
a3b4	a5b6
a2b5	a6b7
a4b5	
a5b6	
a6b7	

a1b2	a1b2
a1b3	a1b3
a1b4	a1b4
a4b5	a4b5

Choose a mapping between members of the collated sets according to the fan-in count.

In cases where multiple matches are possible, a random choice can be made. This might lead to finding local maximas and not the global maxima.

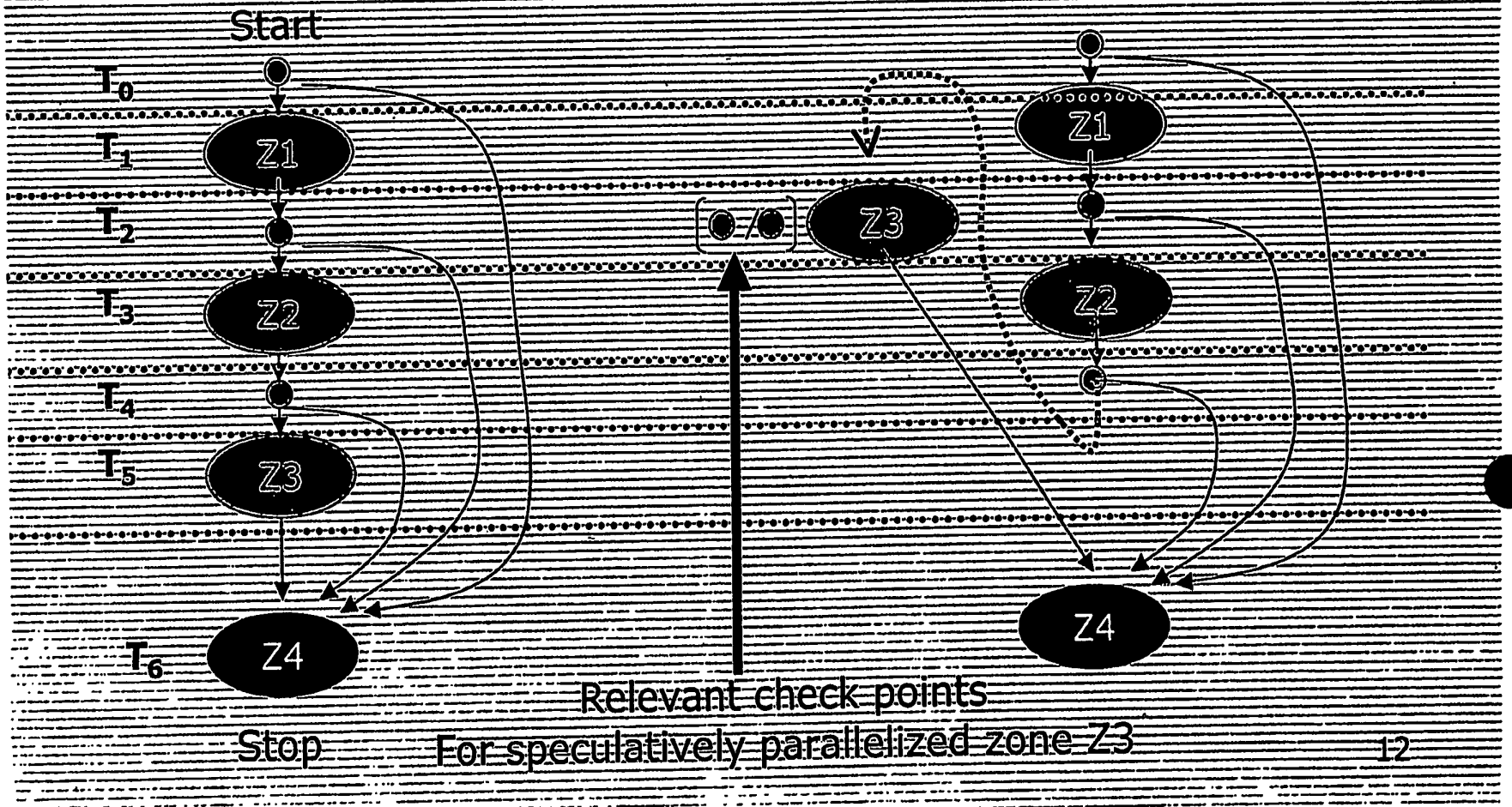
Proposed Approach

To reduce the possibility of getting trapped in local maximas, better choices can be made in the multiple choice cases, based on
(i) fan-in count of the source nodes
(ii) Whether the edge to a node beyond the current node set is a control flow type of data flow type.

By comparing graphs in terms of strings through a gradient ascent approach, a near optimal solution to the Largest Common Sub-graph (LCSG) can be obtained.
This process needs to be done for all possible reconfigurable zones, in order to obtain Clusters (LCSGs).
The candidate graphs at each stage are chosen from alternate paths of hammock like structures in the CDFGs.
Once a Cluster is identified, that region of the CDFG is modified to a coarser granularity. Thus we obtain Clusterized CDFGs

Speculatively Parallelizing CCDFGs

While parallelizing zones based on speculative execution, only data dependency on another zone must prevent further attempt to execute that zone in an earlier possible time. The relevant check points must be stored along with the zone / process.



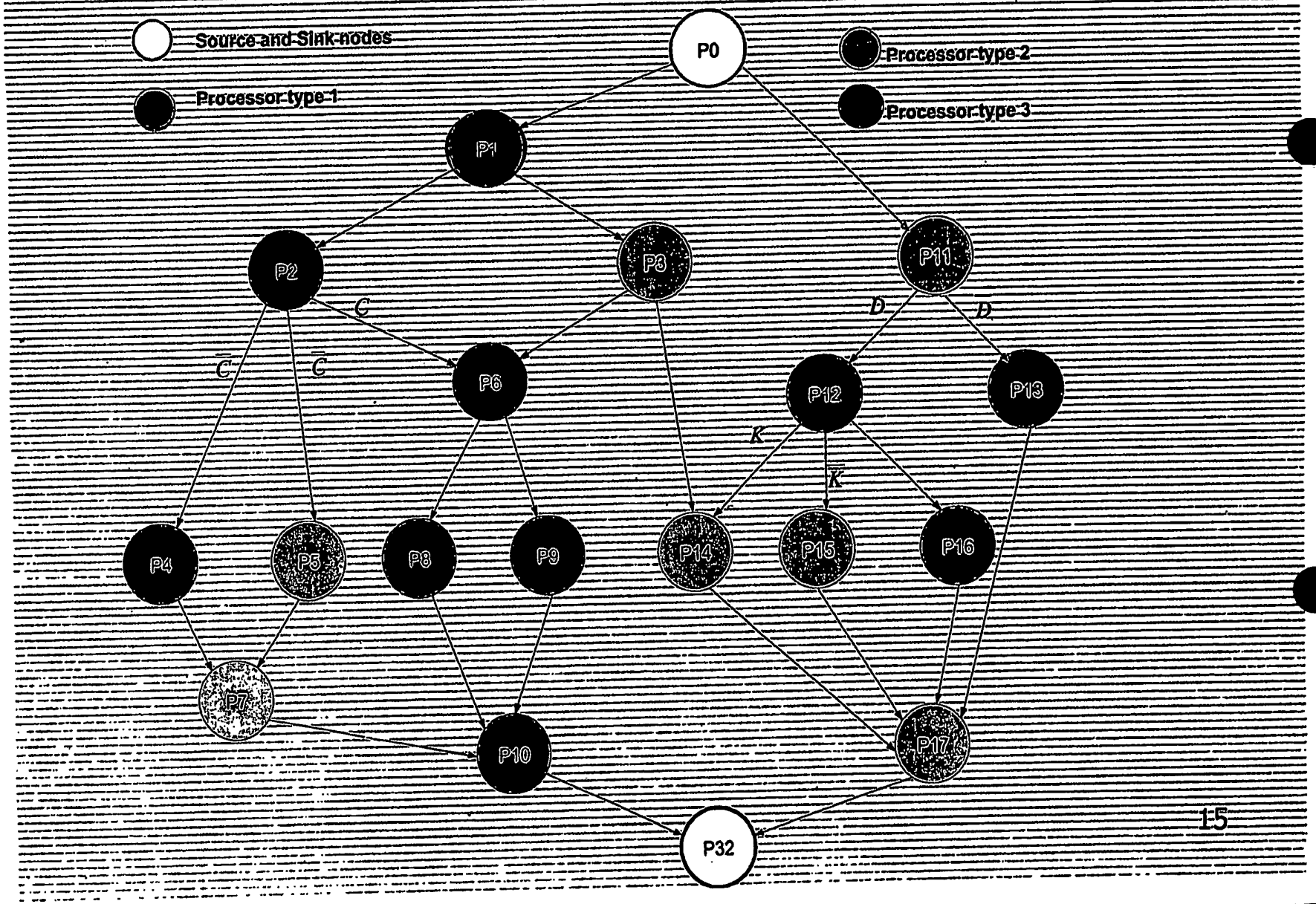
3. Need to obtain Resource count (processing units)

Determining the # of Processors

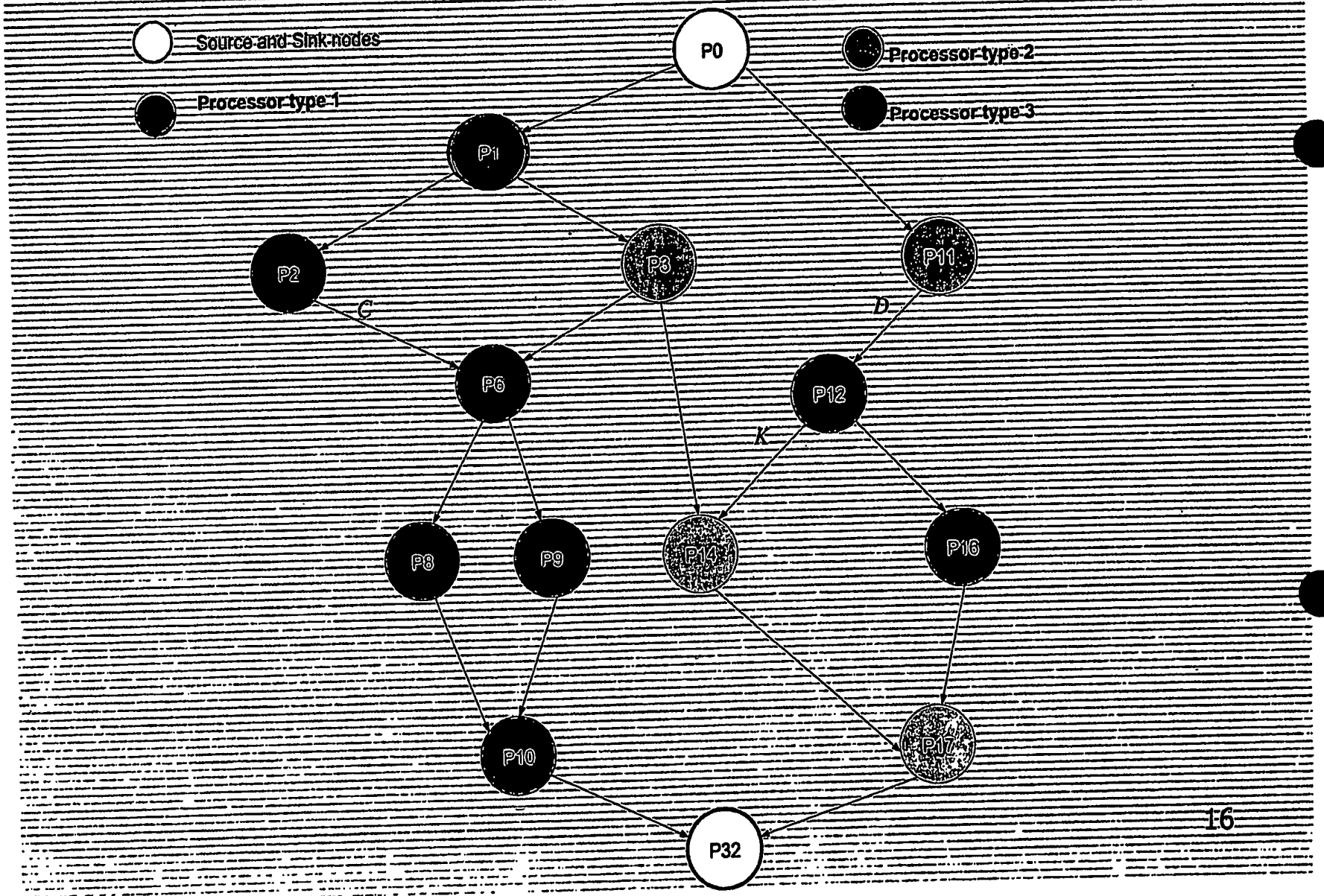
- Obtain sub-graphs for each conditional expression
- Consider only most-likely to occur sub-graphs
  - In case an unlikely path does end up being taken, the clock speed for the general purpose computing resources (the programmable LUTs) must be designed suitably if real-time requirements exist.

Isolate nodes corresponding to a specific processor type in each sub-graph

CCDFG example

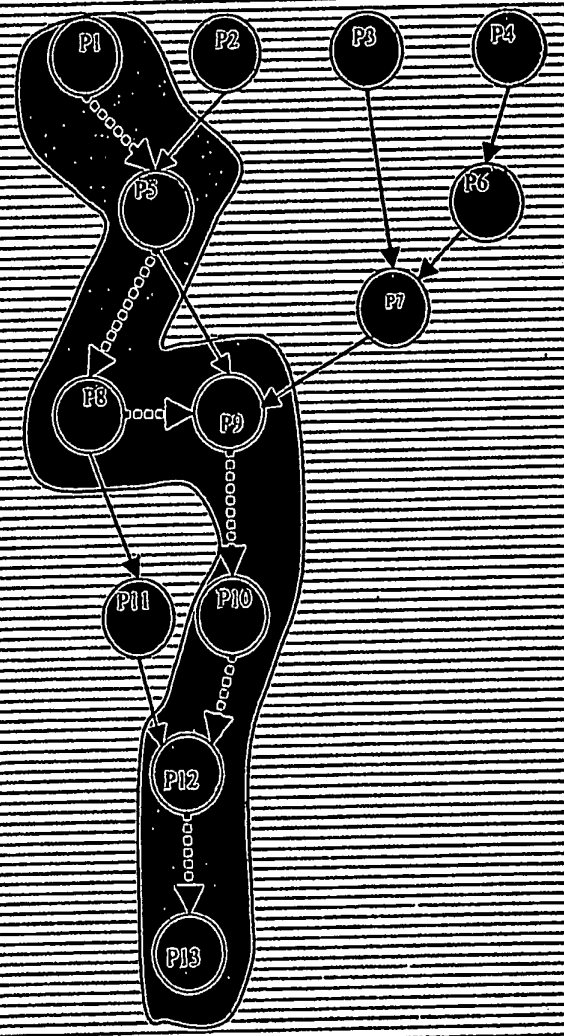


Sub-Graph for DCK



Methodology Contd.

- Segregate the graph into a primary set and secondary set, based on a weighted measure of execution times of nodes, the data address space accessed by nodes and the amount similarity between the graphs representing individual nodes (reconfiguration)
- Allocate a PE for nodes in the primary path.
- Segregate the nodes in the secondary set into a primary set and otherwise
- Perform a step similar to b.
- Continue till as close to a 1:1:1... of resource utilization is achieved.



4. Need to Schedule (map) CDFG on the available resources

Scheduling: Types & Existing Strategies

- **Static: all possible schedules are determined ahead of execution time and stored in a table.**
- **Dynamic: a scheduler manages the task of resource allocation at run time.**

Dynamic vs. Static Scheduling

- Dynamic scheduling

  Pros: Less effort at compile time

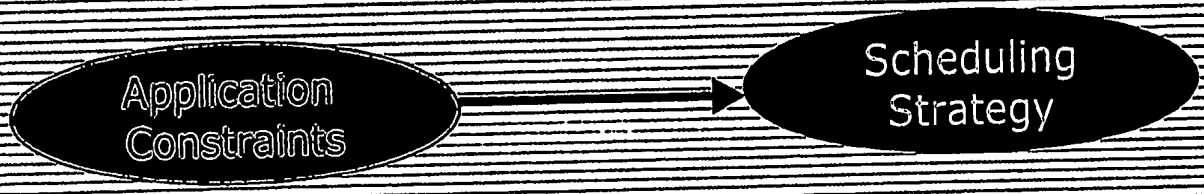
  Cons: Harmful in reconfiguration style processors because of the large communication overhead.

- Static scheduling

  Pros: Optimal solution possible. Guaranteed worst case execution time.

  Cons: Requires tables to be stored on the chip.

Which to Adopt ?

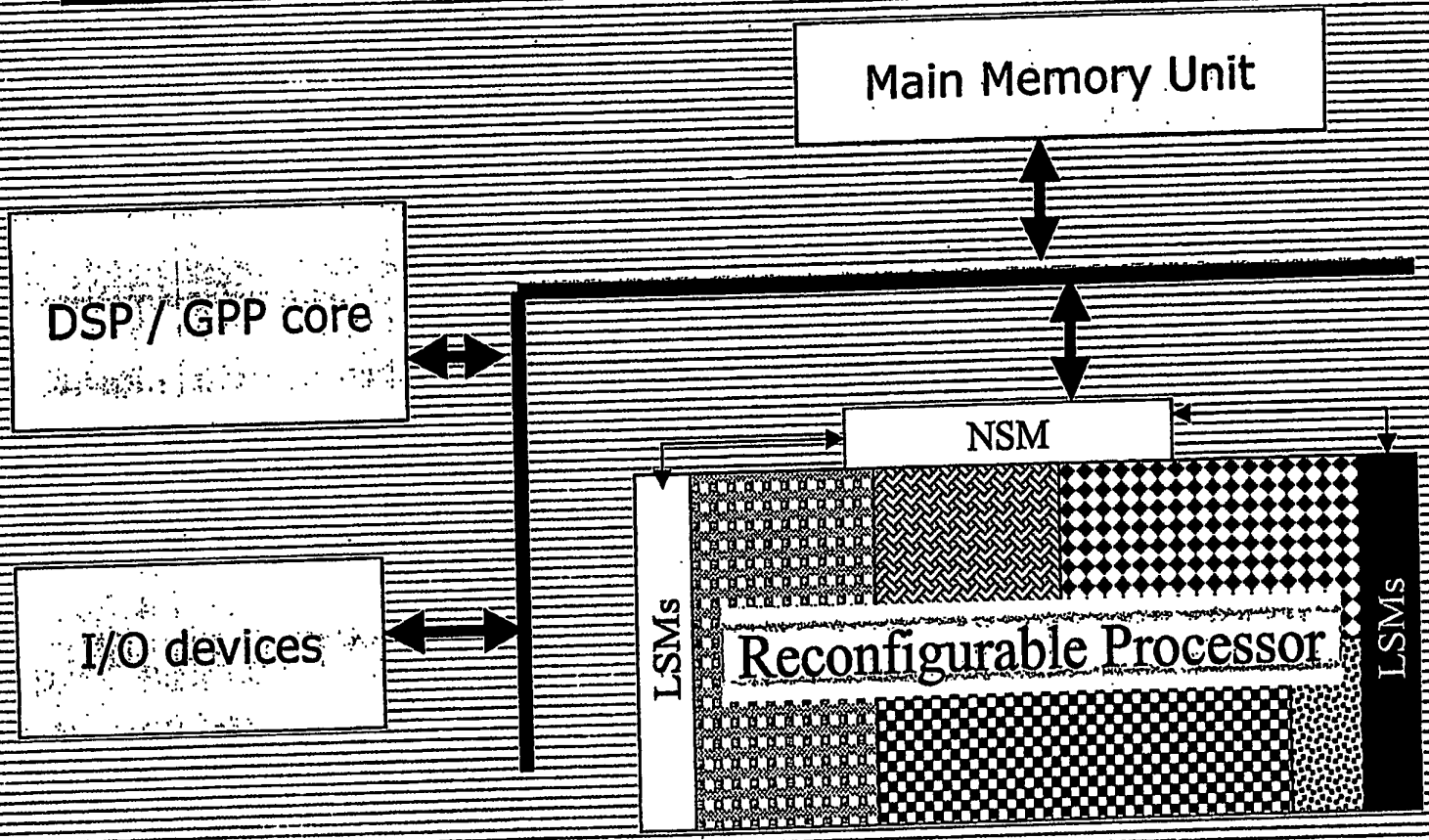


In the case of a *real time application*, where frequent communication on a chip is expensive, static table based scheduling is more feasible (provided only parts of the static tables are loaded at a time with some look ahead).

Key Points in Static Scheduling

- The need to obtain as tight a bound as possible before calling forth the branch & bound approach.
- Loops representing single processes and unknown execution times.
- Loops with Control Structures
- One Loop Feeding Data to Another (Low Memory Footprint)
- Nested Loops with Changing Inner-loop Limit
- Integration of reconfiguration time.

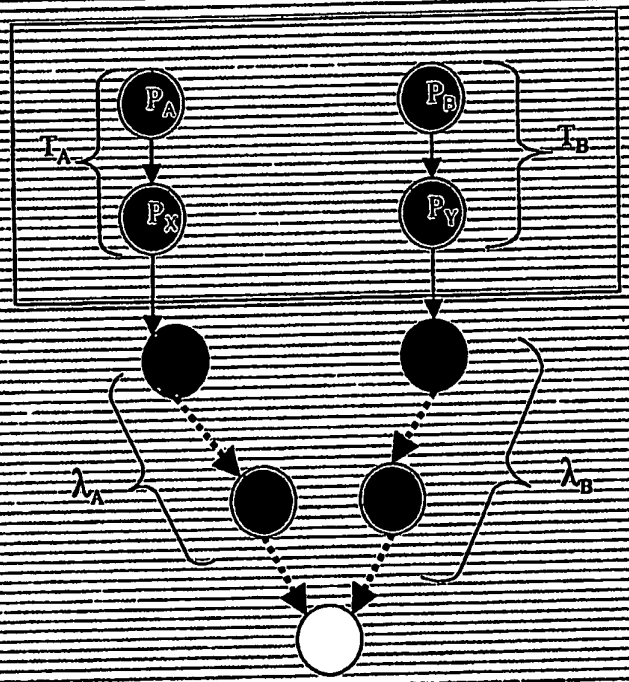
A Peek into our Architectural Overview



LSM = Logic Schedule Manager; NSM = Network Schedule Manager

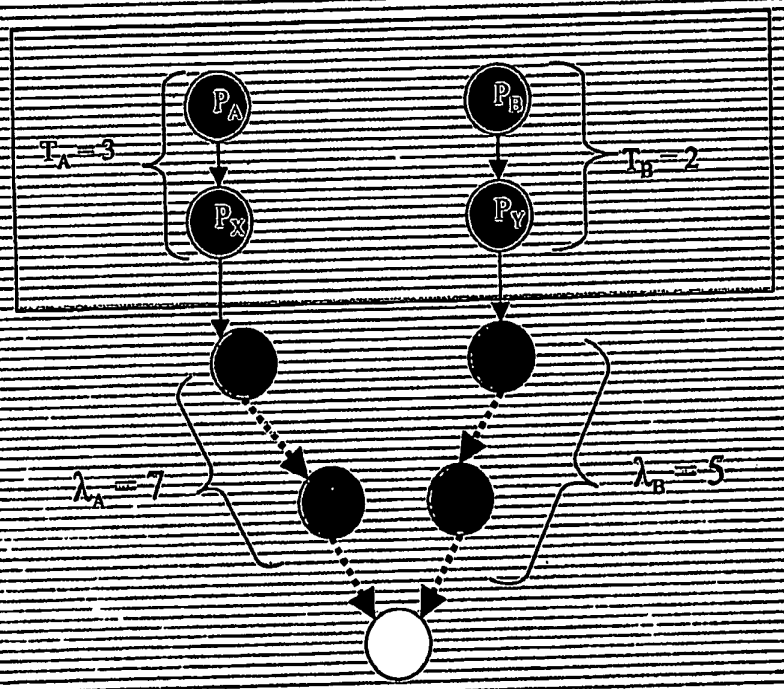
Factor 1: Need for a Tight Bound

PCP



If lambdaA < lambdaB, then PB is scheduled first (Pop).

What if there is a Resource Conflict ?



Need to Improve PCP

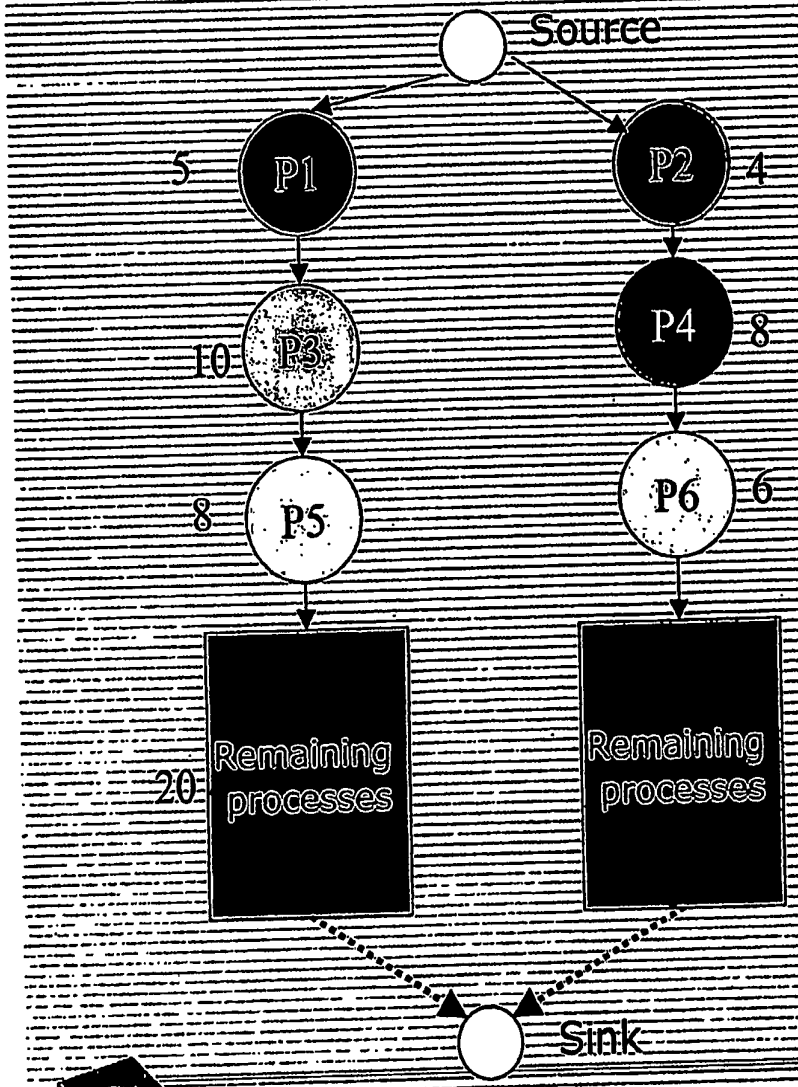
- PCP scheduling does not account for resource conflicts between processes that are not in the ready set

  Solution: All major resource sharing processes must be considered, even those beyond the ready set and the max-min policy used (example in following slide).

- Taking into account every resource conflict is impossible, as the complexity would explode

  Solution: Only the most complex resource conflicts are accounted for using a Modified PCP scheduling algorithm. Other resource conflicts are dealt with in the same way as in PCP scheduling

Improved PCP Example

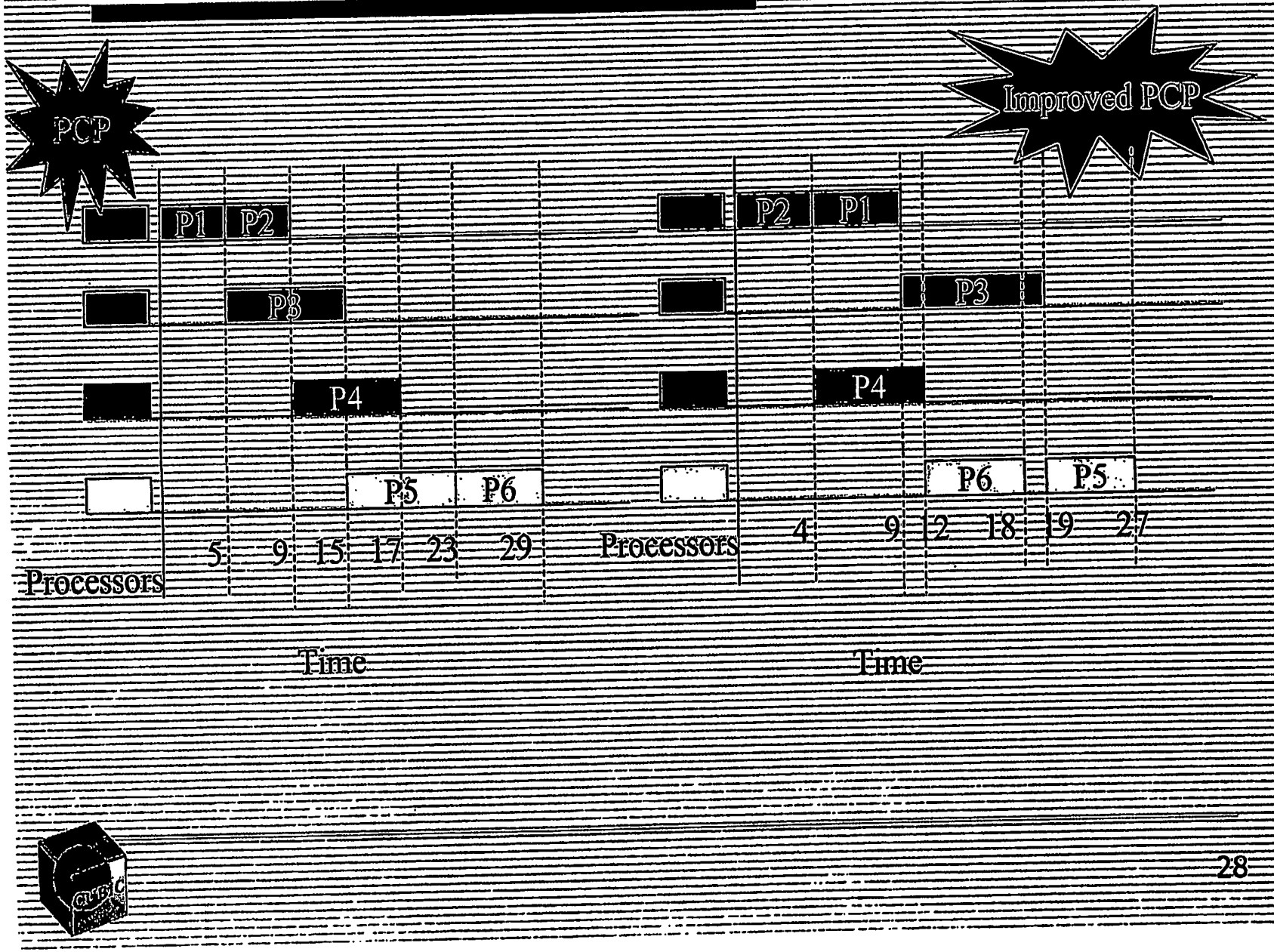


Scheduling Options (Worst-case time):

- P1 → P2 & P5 → P6
  Max(P1P2P4P6, P1P3P5P6) (29 units)
- P1 → P2 & P6 → P5
  Max(P1P2P4P6P5, P1P3P5) (31 units)
- P2 → P1 & P5 → P6
  Max(P2P1P3P5P6, P2P4P6) (33 units)
- P2 → P1 & P6 → P5
  Max(P2P1P3P5, P2P4P6P5) (27 units)

Find the worst-case execution time in each of these options and the minimum time gives the best scheduling option

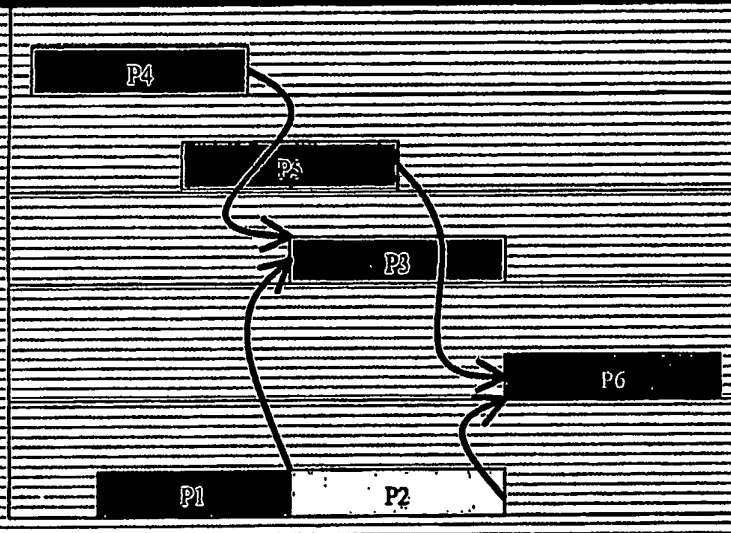
Individual schedules



Process of Using a Bound

- Reduce the CDFG into individual DFG's
- Use improved PCP algorithm to determine the near optimal schedule for each DFG
- Arrange the DFG's in the decreasing order of their execution times
- Generate the Branch and Bound schedule for the worst-case DFG
- For other DFGs, generate BB schedule *iff*
  - Improved_PCP_delay >= Worst_BB_delay
- Merge all the final schedules towards the worst case.

Factor 2: Loops with Unknown Execution Times



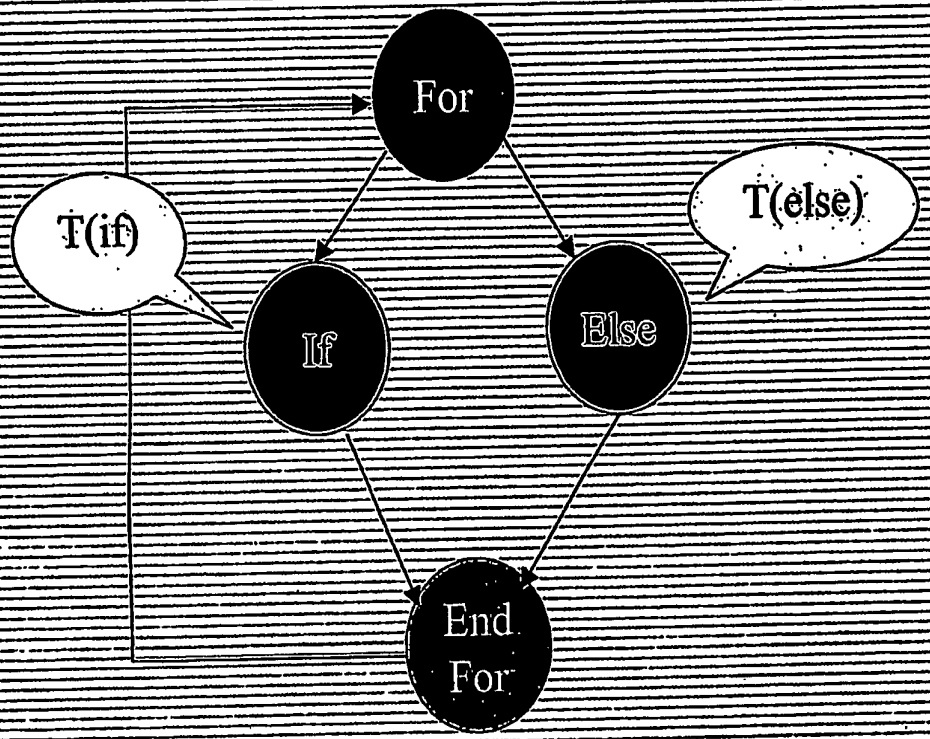
Solution:

- In case of prediction failure, all dependent processing units should be notified

Assumptions

- ✓ **Most probable case from trace file statistics**
- ✓ **All dependent schedules are given a small buffer time before they start, to accommodate for notification in case prediction fails**

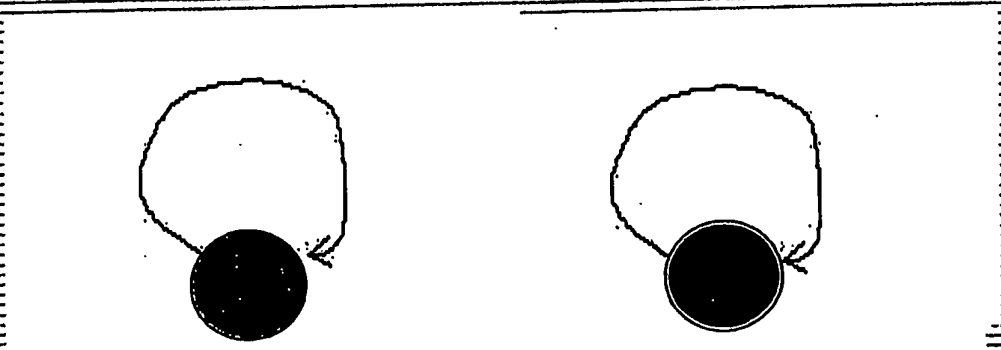
Factor 3: Loops with Control Structures



Solution:

- The loop is scheduled with the worst of the 2 cases execution time ⇒ MAX(T(if),T(else))
- The loop is scheduled for most probable number of iterations. The schedule after the loop can be pulled / pushed during run time

Factor 4: Low Foot-print Iterations



	Expression-α	Expression-β	Expression ?	Expression ?
Process-A-(10)		0		
Process-B-(20)		10		
.....				
.....				

Solution: Dynamic Table Updating

	Expression-α	Expression-β	Expression ?	Expression ?
Process-A-(10)		20		
Process-B-(20)		30		
.....				
.....				

.....and-so-on.

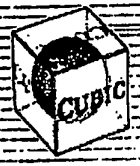
Factor 5: Nested Loops with Changing Inner-loop Limit

```
For I = 1 to N do
{
    For J = 1 to I do
    {
        Processes
    }
}
```

N is not known during compile-time

Solution:

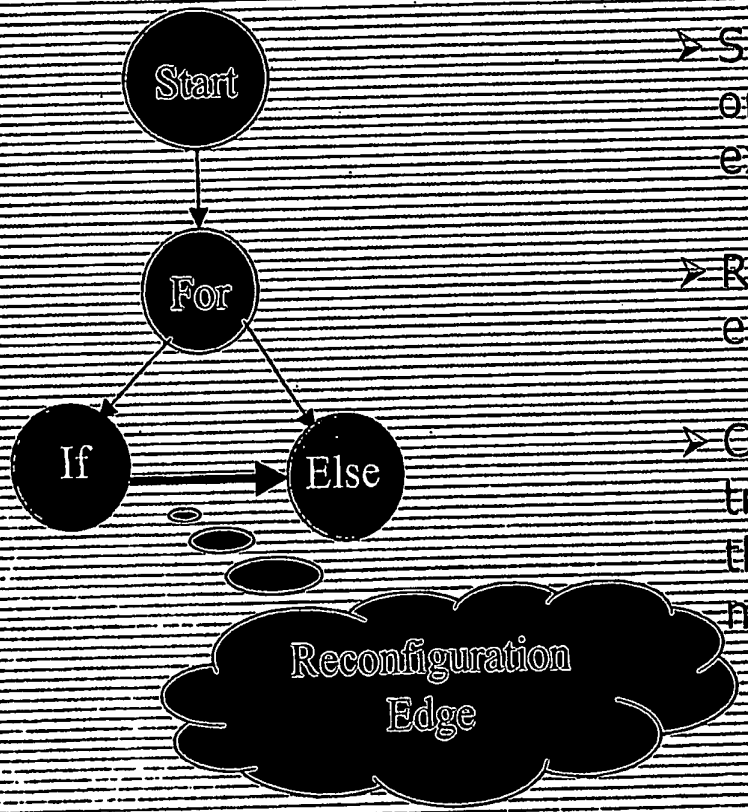
- The worst-case estimate of N is calculated
- Schedules for inner loop for each of the iteration of the outer loop are determined
- The NSM is initialized with the inner loop schedule for worst-case of N
- The NSM is then updated with the other schedules as the outer loop iteration count changes



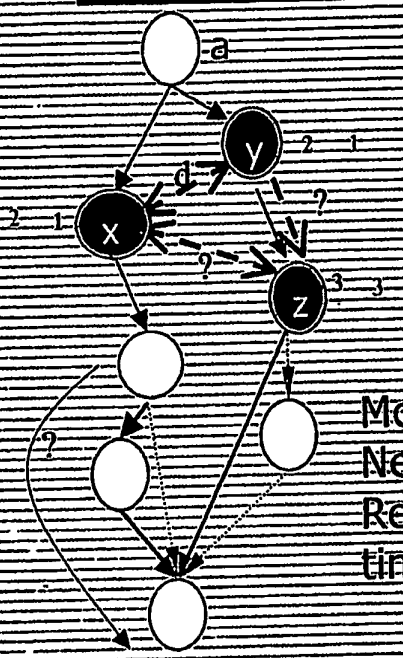
Factor 6: Reconfiguration Time

Solution:

- Special reconfiguration edges are added onto the control flow graphs. These edges exist between a similar set of processes
- Reconfiguration times affect the worst-case execution time of loopy codes
- Care needs to be taken to schedule the transfer of reconfiguration bit-stream from the main memory to the processor memory.



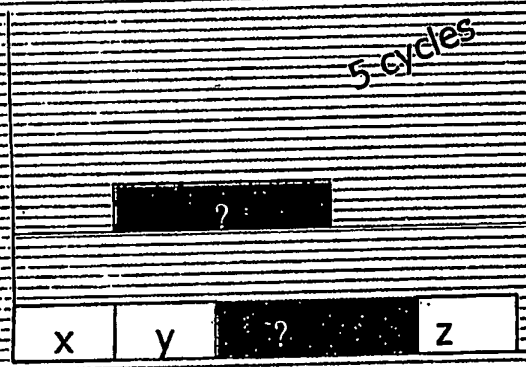
Simple Example of Solution 6



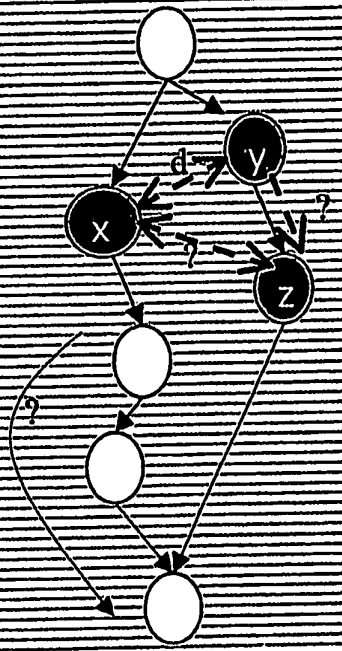
If a = True => green
If a = False => red

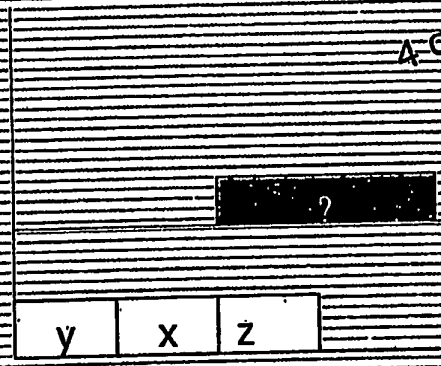
d = 0
? = 2
? = 0

Mooney's scheduler
Neglecting
Reconfiguration
time



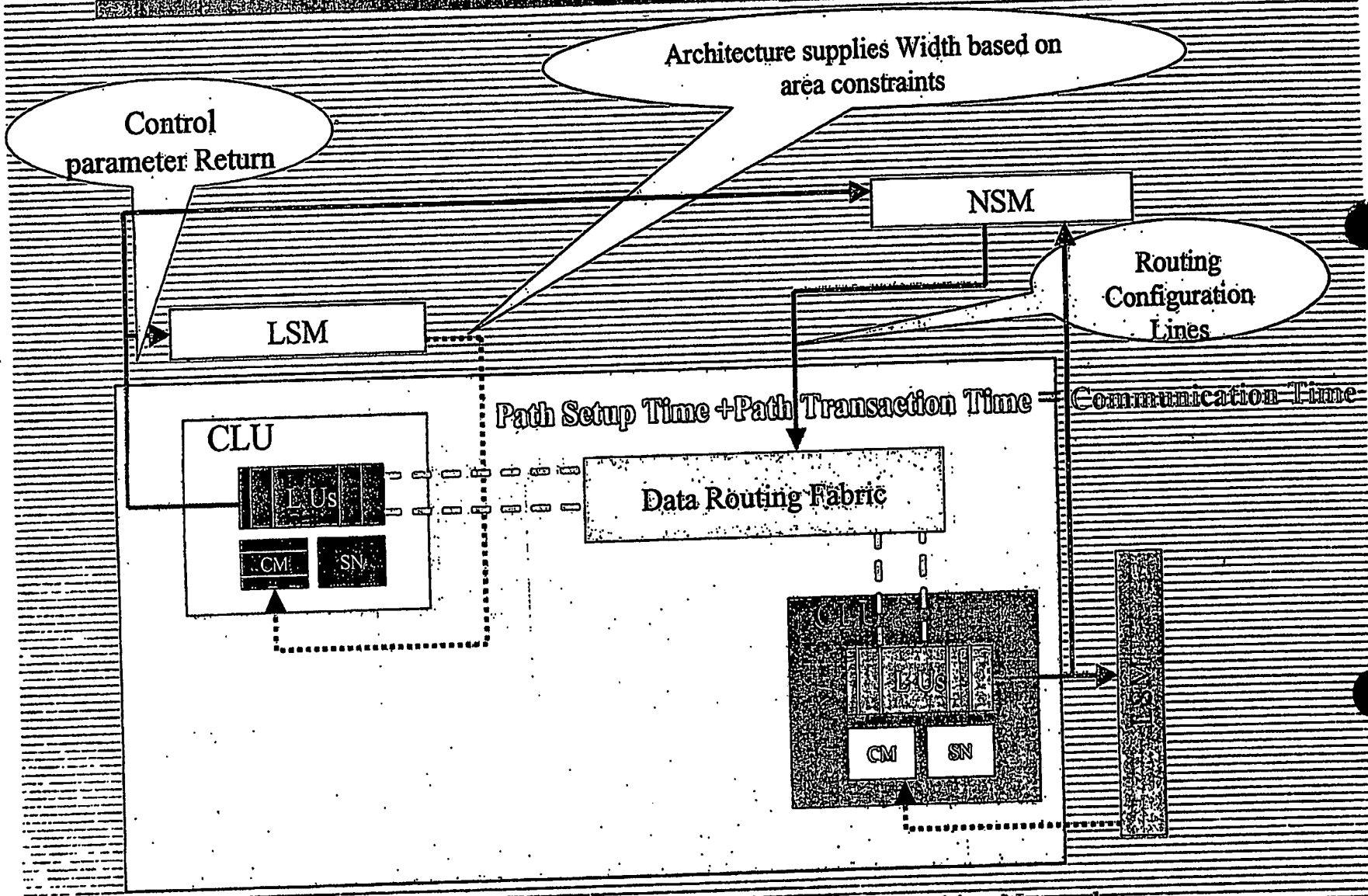
Considering
Reconfiguration
time





- **Overview of Architecture**

Architectural Overview



CLU = Configurable Logic Unit; LU = Logic Units; SN = Switching Network
CM = Configuration Memory; LSM = Logic Schedule Manager

Architectural Issues

- The Network Schedule Manager has access to a set of tables, one for each CLU.

A table consists of possible tentative schedules for processes or tasks that must be mapped onto the corresponding CLU subject to evaluation of certain conditional control variables.

- The Logic Schedule manager schedules and loads the configurations for the processes that need to be scheduled on the corresponding CLU ie. all processes that come in the same column (a particular condition) in the schedule table.

Architectural Issues

- Assuming that once a configuration has been loaded into the CM the process of using it is instantaneous, its always advantageous to load successive configurations into the CM ahead of time. This will mean a useful latency hiding for loading a successive configuration.

The context for the processes are tracked by the NSM and LSM.

- The reconfiguration time is dependent on two factors:
  - How much configuration data needs to be loaded into the CM (Application dependent)
  - How many wires are there to carry this info from the LSM to the CM (Architecture dependent)

Path set up time will be a part of the communication time and not explicitly considered as reconfiguration time.

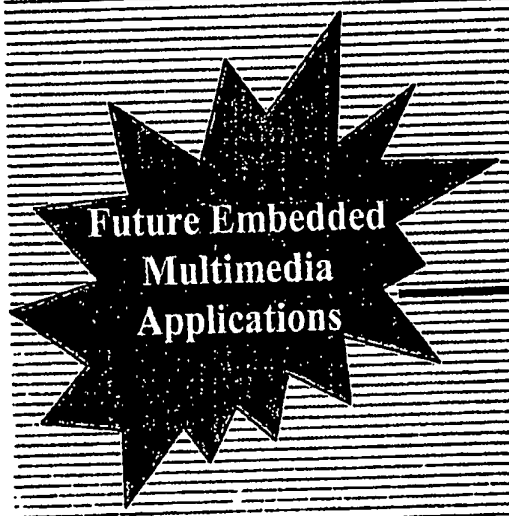
Architectural Issues

- The Network Schedule Manager should accept control parameters from all LSMs.

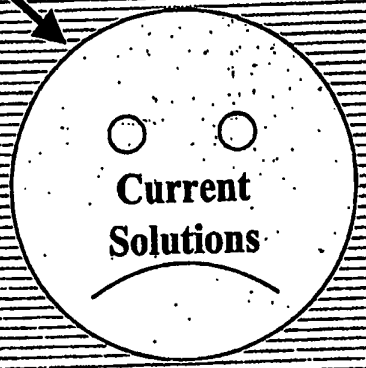
  It should have a set of address decoders because to send the configuration bits to the Network fabric consisting of a variety of switch boxes, it needs to identify their location.

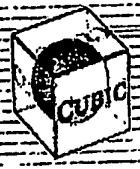
- Therefore for every column in the table, its needs to know the route apriori (SHOULD NOT try to find the shortest path at run time) For a given set of processors communicating, there should be a fixed route. If this is not done then, the the communication time of the edges in the CCDFG cannot be used as constants while scheduling the graph.

Conclusion



- Dynamic application swaps
- Longer Hardware Shelf life
- Quicker Design Cycles





ARTIFACT SHEET

Enter artifact number below. Artifact number is application number + artifact type code (see list below) + sequential letter (A, B, C ...). The first artifact folder for an artifact type receives the letter A, the second B, etc..
Examples: 59123456PA, 59123456PB, 59123456ZA, 59123456ZB

60/523462ZA

Indicate quantity of a single type of artifact received but not scanned. Create individual artifact folder/box and artifact number for each Artifact Type.

- ☐ CD(s) containing computer program listing
  Doc Code: Computer Artifact Type Code: P
- ☐ Stapled Set(s) of Extra Color Drawings/Photographs
  Doc Code: Artifact Artifact Type Code: C
- ☐ CD(s) containing pages of specification ☐ and/or sequence listing
  Doc Code: Artifact Artifact Type Code: S
- ☐ CD(s) with content unspecified
  Doc Code: Artifact Artifact Type Code: U
- ☐ Microfilm(s)
  Doc Code: Artifact Artifact Type Code: F
- ☐ Video tape(s)
  Doc Code: Artifact Artifact Type Code: V
- ☐ Model(s)
  Doc Code: Artifact Artifact Type Code: M
- ☐ Bound Document(s)
  Doc Code: Artifact Artifact Type Code: B
- ☑ Other, description: Appendix
  Doc Code: Artifact Artifact Type Code: Z